P^2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling

Bo Feng
Northeastern University
feng.bo@husky.neu.edu

Alejandro Mera
Northeastern University
mera.a@husky.neu.edu

Long Lu
Northeastern University
l.lu@northeastern.edu

Abstract

Dynamic testing or fuzzing of embedded firmware is severely limited by hardware-dependence and poor scalability, partly contributing to the widespread vulnerable IoT devices. We propose a software framework that continuously executes a given firmware binary while channeling inputs from an off-the-shelf fuzzer, enabling hardware-independent and scalable firmware testing. Our framework, using a novel technique called P^2IM, abstracts diverse peripherals and handles firmware I/O on the fly based on automatically generated models. P^2IM is oblivious to peripheral designs and generic to firmware implementations, and therefore, applicable to a wide range of embedded devices. We evaluated our framework using 70 sample firmware and 10 firmware from real devices, including a drone, a robot, and a PLC. It successfully executed 79% of the sample firmware without any manual assistance. We also performed a limited fuzzing test on the real firmware, which unveiled 7 unique unknown bugs.

1 Introduction

Microcontrollers, or MCU, are commonly used for building IoT (Internet of Things) and modern embedded devices, thanks to their high energy-efficiency, extensible connectivity, and adequate computing power. As MCU devices become widely deployed in various scenarios, ranging from smart homes to industrial systems, their security has been raised as a major concern among users and operators. As demonstrated in recent reports [44], software vulnerabilities cause the majority of attacks on MCU devices, resulting in not only digital but also physical damages.

MCU firmware (i.e., the whole software stack on MCU) contains vulnerabilities just as computer software does. Most MCU vulnerabilities are virtually the same in nature as their computer counterparts. Therefore, it would be ideal if the proven vulnerability discovering techniques on computers, such as fuzz-testing or fuzzing, can be applied to MCU firmware. However, in reality, off-the-shelf fuzzers cannot directly test firmware, which partly contributed to the fact that many firmware is not sufficiently tested for security vulnerabilities [26].

This work was supported by the National Science Foundation (Grant#: CNS-1748334), the Office of Naval Research (Grant#: N00014-17-1-2227), and the Army Research Office (Grant#: W911NF-17-1-0039). The extended version of the paper with more details can be found at https://arxiv.org/abs/1909.06472.

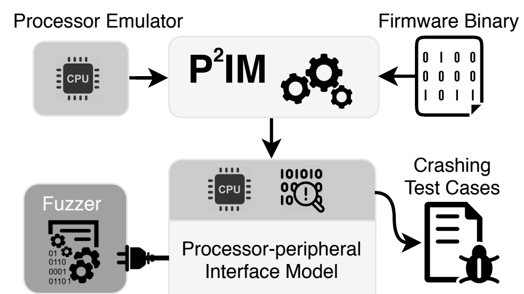


Figure 1: Framework Overview

The inapplicability of fuzzers on MCU boils down to the lack of a platform where firmware can execute while taking inputs from fuzzers. Existing emulators cannot help because none of them emulates the whole range of MCU peripherals (i.e., unable to run firmware). Some recent works addressed this issue using hybrid emulation [31, 40, 48, 51], which forwards peripheral operations to real devices. Although this approach creates a platform to run firmware with a fuzzer, the platform is fairly slow and can hardly scale due to the hardware dependence.

We present a novel approach to firmware fuzzing. We design a framework to run and test MCU firmware at scale without any hardware dependence. Our framework takes a firmware binary as input and hosts an unmodified fuzzer (AFL [52]) as a drop-in component. Using a generic processor emulator (QEMU), the framework executes the firmware and handles its peripheral accesses while channeling the fuzzing input and feedback between the firmware and the fuzzer. Figure 1 shows an overview of the framework.

The key technique used in our framework is called P^2IM (or Processor-Peripheral Interface Modeling). It automatically

models the I/O behaviors of a wide range of peripherals while treating peripherals themselves as black boxes. The generated models satisfy a property we formulated, called Processor-Peripheral Interface Equivalence. We show that when this property is satisfied, an emulated execution of firmware can continue smoothly (e.g., no crash, stall or skipped peripheral operations) without requiring any dependent peripherals (either real or emulated).

We evaluated our framework using 70 sample firmware and 10 firmware from real MCU-based devices, including a drone, a robot, and a PLC (Programmable Logic Controller). The result shows that our framework can continuously run 79% of the sample firmware without any crash, stall or skipped peripheral operations. We also performed basic fuzzing (without memory sanitizer) on the real firmware and discovered 7 unique and previously unknown bugs.

2 Roadmap & Overview

2.1 MCU Firmware & Testing

Firmware generally means any low-level software that controls hardware in a computing device. In this paper, we focus on firmware for microcontrollers (MCU). These devices are cost- and power-effective computers built for specific purposes, such as the motion controller of a self-balancing robot, the engine control units (ECU) in a car, etc. STM32L010F4 [22] is an example of ultra-low-power MCUs commonly used in IoT devices. It carries an ARM Cortex-M0+ processor at 32MHz, 16KB flash as the persistent storage, 2KB of RAM, and a wide range of peripherals.

MCU firmware is usually a monolithic piece of software that contains peripheral device drivers, a tiny OS or system library, and a set of specialized logics or applications. For example, the firmware in a MCU-based drone contains the drivers for all onboard peripherals, either a small real-time operating system or vendor-customized system library, and the PID (proportional, integral and derivative) controller among other application-level logics. We note that MCU vendors rarely use general-purpose OS such as Linux to build MCU firmware. Due to hardware constraints, they prefer an OS specifically designed for MCU or simply use a thin system library in lieu of a stand-alone OS (called bare-metal devices). In the rest of the paper, we refer to MCU firmware simply as firmware for brevity.

Due to the fast development and wide adoption of MCU devices in cyber-physical and IoT systems, the security issues of these devices, often caused by vulnerable firmware [44], have led to severe consequences and become a major concern among users and operators [25]. To mitigate vulnerable MCU devices, researchers recently proposed techniques for fuzz-testing firmware [31, 40, 51]. These techniques allow partial execution of firmware on an emulator while forwarding unsupported operations (e.g., peripheral I/O) to real hardware.

This line of work allowed fuzzing to be applied to firmware. But due to the hardware dependence and slow forwarding, fuzzing through these partial emulators can hardly scale up. For instance, the number of parallel fuzzing runs is limited by the availability and capacity of the dependent hardware; the speed of each fuzzing run is severely capped by the I/O forwarding, which is three orders of magnitude slower than native I/O [40]. As a result, high scalability, the key requirement for effective software fuzzing, cannot be achieved when using partial emulation that depends on slow and limited hardware.

2.2 Open Challenges

If the state-of-the-art fuzzers could work directly on firmware at scale, the significant values of these fuzzers demonstrated on computer software (e.g., unparalleled vulnerability discovery ability) can automatically transfer to MCU firmware, which can tremendously help reduce vulnerabilities and improve security of MCU devices. However, despite the previous efforts aiming at this goal [31, 40, 48, 51], we still identified the following open challenges that prevent fuzzers for computer software from being effective on firmware.

Hardware Dependence: Previous efforts on firmware fuzzing require certain hardware (e.g., peripherals). This is due to incomplete hardware emulation. Moreover, such dependent hardware is much slower than emulators running on computers. As a result, the hardware dependence introduces orders of magnitudes of delays to fuzzer execution. Moreover, hardware dependence also critically limits parallelism. For instance, one dependent peripheral can only be used by one fuzzing session. Therefore, highly parallel fuzzing, which is the key to fuzzers' success on computer software, is not achievable.

Wide Range of Peripherals: Due to the poor performance and scalability caused by hardware dependence, some recent work proposed purely emulation-based fuzzing of firmware. In fact, fuzzing software on a fully emulated platform has been found useful for a long time in cases where software under test cannot be instrumented or is only available in binary forms. However, creating fully emulated MCU has proven impractical and no existing emulators offer generic MCU support. This is mainly because of the highly heterogeneous MCU hardware in general and the wide range of peripherals in particular. Each firmware may interact with a distinct set of peripherals, which can be customized by the MCU vendor. Peripherals of the same type but different models/brands often have different specifications and interfaces. Therefore, a specially customized emulator is often required for fuzzing or testing a new firmware. Building such emulators remains a manual task, which is not only error-prone but impossible to catch up with the large and fast-increasing number of MCU devices.

Diverse OS/System Designs: In addition to the hardware-related challenges unique to MCU, the software also poses challenges to firmware fuzzing that are currently unaddressed. Unlike general-purpose computers, whose

OSes are dominated by a few mainstream options that follow similar designs, MCU devices use a much larger and more diverse set of OSes that are significantly different from each other. Many MCU devices do not even have a typical OS but a system library that manages hardware and task scheduling. The diverse OS/system designs among MCU means that OS-specific fuzzing methods, which existing system fuzzers use (e.g., syscall fuzzers), are not applicable to firmware. In other words, generic firmware fuzzing should be OS-agnostic and not make assumptions about the OS/system designs.

Incompatible Fuzzing Interfaces: Another software-related challenge unique to firmware fuzzing is about the interfaces through which fuzzer-generated inputs are channeled into firmware execution. For computer software fuzzing, the input interfaces are well-defined and uniform (e.g., via files or standard I/O). However, firmware reads all inputs via peripherals, which come in many different types and have their own access conventions. Making the matter more complicated, different drivers in firmware may configure the same peripheral differently and then perform I/O through different interfaces. As a result, the input interfaces supported by existing fuzzers are incompatible with firmware. Moreover, manually adding support for every peripheral I/O interface to fuzzers can be a daunting task, if possible at all.

We note that the aforementioned challenges are unique to MCU firmware fuzzing. There are other open problems facing software fuzzing in general, such as better input generation, more effective error detection, etc. However, this work is focused on tackling the challenges unique to firmware fuzzing. We consider improving general fuzzing techniques orthogonal and out-of-scope for this paper.

2.3 Our Approach

We present a novel approach to MCU firmware fuzzing, which overcomes the challenges discussed before. We design a framework that supports fuzzers as drop-in components to test firmware in a scalable and hardware-independent fashion. The framework aims to solve the MCU-imposed fuzzing challenges while allowing fuzzers to focus on performing and improving their own job (i.e., generating inputs and finding bugs). The goal of our framework is to bridge the wide open gap between fuzzers and firmware. It allows existing fuzzers to test firmware without any knowledge about the software and hardware design of MCU. It also facilitates the development of specialized fuzzers for firmware.

Our approach is novel in that it neither relies on any hardware nor emulates peripherals. We introduce a form of *approximate MCU emulation* for supporting firmware testing and fuzzing. More importantly, we provide a method to automatically generate approximate emulators based on firmware binaries. The approach is inspired by our observation that firmware can execute on an emulator without real or fully emulated peripherals, as long as the emulator provides the firmware with

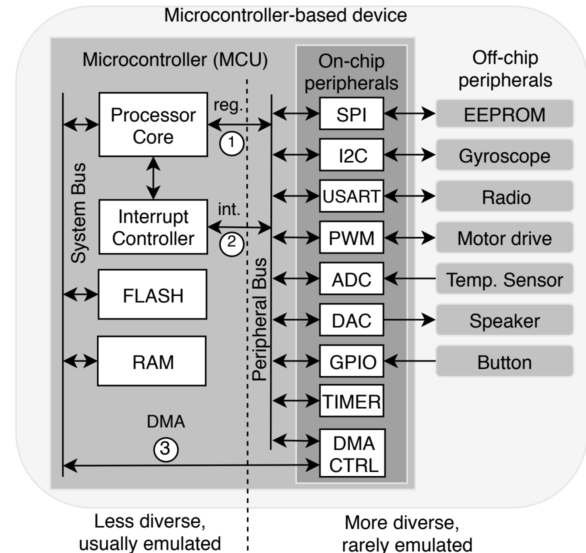


Figure 2: Architecture diagram of MCU devices. Firmware running on processor core interacts with peripherals via ① memory-mapped registers, ② interrupts, and ③ DMA.

acceptable inputs from peripherals when needed. Such inputs do not have to be the same as what a real peripheral would produce. But they do need to pass firmware's internal checks to avoid disrupting firmware execution. For a given firmware, our approximate emulator uses a generic processor/ISA emulator (e.g., one for ARM Cortex-M) and a model, automatically built for the firmware, that captures what constitutes an acceptable input for each peripheral accessed by the firmware. We find that this kind of approximate emulation can comprehensively exercise a firmware (i.e., covering most firmware code), and therefore, is sufficient for supporting fuzzing and other types of firmware analyses that examine the control or data behaviors of firmware (as opposed to testing functional correctness), such as taint analysis, invariant detection, etc.

Next, we provide the necessary background on peripheral interfaces. We then define a property that an approximate MCU emulation must meet in order to be acceptable for supporting firmware fuzzing. At the end of the section, we discuss the high-level design of our framework that enables the approximate firmware execution and supports firmware fuzzing.

2.4 Processor-Peripheral Interfaces

Peripherals are indispensable from MCU devices and of great varieties. As shown in Figure 2, they can be on-chip or off-chip. On-chip peripherals typically serve as the proxy through which data travels between firmware and off-chip peripherals. Some on-chip peripherals are not externally connected and provide simple functionalities needed by firmware (e.g., timers). In this paper, we only consider on-chip peripherals (or peripherals for short) because firmware cannot access off-chip peripherals directly. As illustrated in Figure 2, there are three types of peripheral I/O interfaces

exposed to firmware, namely memory-mapped registers, interrupts, and direct memory access (DMA). Firmware performs all I/O through these interfaces. We refer to the first two types (① and ② in Figure 2) as *processor-peripheral interfaces* as they connect processors and peripherals.

We note that this work covers the processor-peripheral interfaces and not DMA. We leave DMA out of scope for this paper because it is extremely difficult to model automatically and its I/O behavior is heavily dependent on internal designs of individual peripherals, which our method is oblivious of to be generic. Nonetheless, DMA is not frequently used by MCU peripherals, which tend to exchange small amounts of data with firmware (only 2 out of 70 firmware tested in §5.1 use DMA).

We define a new property, related to peripheral I/O modeling, for MCU emulators. It is called Processor-Peripheral Interface Equivalence (or P^{2}IE). Satisfying this property means that: (1) the emulator emulates the *processor-peripheral interfaces*, rather than peripherals themselves used by the firmware, and (2) the emulated interfaces are equivalent to those of the peripherals expected by the firmware, in terms of their impact to firmware execution. The formulation of P^{2}IE is based on our experience with firmware analysis for a wide range of MCUs. We observed that providing equivalent processor-peripheral interfaces is sufficient for a generic emulator, without any peripheral emulation, to comprehensively execute and test/fuzz firmware. A P^{2}IE-enabled emulator handles peripheral I/O operations by providing the processor-peripheral interfaces and mimicking their external behaviors.

We also define an empirical test for P^{2}IE: the property is satisfied if the firmware running on the peripheral-agnostic emulator never *crashes, stalls, or skips operations* due to peripheral I/O errors. A *crash* may happen when the firmware tries to read/write data from/to a peripheral but encounters a fatal error, such as illegal memory access or unsupported peripheral operations. A *stall* may occur when the firmware waits for a peripheral state to change but the emulator fails to recognize and handle it. Under a similar situation, the firmware may eventually give up on waiting and *skip operations*, causing parts of firmware code to be unreachable. For instance, before reading data from a peripheral such as ADC (analog-to-digital converter), firmware needs to wait for a memory-mapped register bit to be set, which indicates data is ready. If the emulator fails to set such bits, firmware stalls without showing any signs of errors. Alternatively, after a long wait, the firmware simply skips the operations (not only the input operation but also subsequent operations depending on the input).

We use the definition of P^{2}IE to guide our framework design. We use the empirical test of P^{2}IE as a way to verify if our framework generates emulators that satisfy this property.

By focusing on the interface equivalence (i.e., generalizable), rather than emulating every peripheral (i.e., non-generalizable), we demonstrate that it is *possible to automatically build approximate emulators* for MCU devices equipped with a wide range of peripherals. This automated generation of MCU emulators is the key to hardware-independent, scalable, and high-coverage firmware testing.

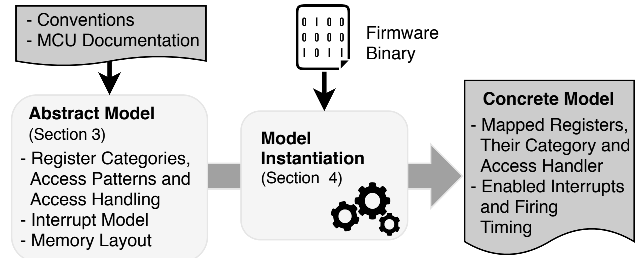


Figure 3: P^{2}IM workflow

2.5 Framework Overview

The framework, for the first time, allows firmware to be dynamically tested and fuzzed without using any MCU devices, hardware peripherals, or human assistance.

The model derivation process, called Processor-Peripheral Interface Modeling (or P^{2}IM), contains two steps, as shown in Figure 3. First, an abstract model is defined for a broad class of MCU architectures (e.g., ARM Cortex-M). An abstract model captures the generic patterns and conventions that firmware follows and acceptable input when accessing processor-peripheral interfaces. Such information is readily available in MCU device datasheets or processor documentation. An abstract model also contains a customizable interrupt firing strategy suitable for the entire MCU architecture class. Defining an abstract model is a manual and offline process done by domain experts. It is practical because it only happens once for each class of MCU architectures (only a few architecture classes are common among MCU) and, using our template, defining an abstract model for a new architecture class does not require too much effort. Abstract models do not vary much across different architecture classes. We discuss the definition of the abstract model for ARM Cortex-M, serving as a template for other MCU architectures, in §3.

The second step is model instantiation, which is fully automatic and needed for every firmware to be tested. It instantiates the abstract model defined for the MCU architecture of a given firmware. It concretizes the abstract model with the firmware-specific information, such as where specifically each peripheral register is mapped in memory and what interdependency among the registers, if broken, may impact firmware execution. This firmware-specific (or device-specific) information is necessary due to the high heterogeneity of MCU (e.g., devices using the same architecture often have different peripheral and interface specifications). Without this information, emulators cannot provide the processor-peripheral interfaces equivalent to the real ones (i.e., achieving P^{2}IE). Our framework automatically infers the firmware-specific information using a technique called explorative firmware executions. An instantiated model tells the emulator what con-

stitutes access to peripheral interfaces and how such access should be handled based on its type and the runtime condition. We discuss the details about model instantiation in §4.

Besides P²IM, another important part of our framework design is the support of fuzzers as drop-in components and the feeding of fuzzing inputs to the firmware execution. Our framework does not have special requirements for fuzzers, which are simply treated as black-box input generators. The framework channels the fuzzing input into the peripheral interface access handlers in the emulator, which feed the fuzzing input when the firmware expects raw input data (as opposed metadata or status input) from peripherals. Our framework also provides standard coverage feedback to fuzzers, collected through the emulator. We discuss the fuzzer setup and the fuzzing results on real firmware in §5.4

3 Abstract Model Definition

As illustrated in Figure 3, the first step of modeling the processor-peripheral interfaces is to build an abstract model for a target MCU architecture class. This is the only manual step in P²IM and should be fairly straightforward for embedded system engineers or security researchers with basic knowledge about MCU firmware and hardware. An abstract model captures the generic patterns and conventions that firmware follows when accessing the processor-peripheral interfaces. For instance, firmware for MCU devices based on ARM Cortex-M typically access different types of peripheral registers via memory-mapped I/O (i.e., peripheral registers are mapped to a fixed region in memory for firmware to read-/write). Firmware for these devices also enables a range of peripheral interrupts mainly for performing asynchronous I/O.

We define an abstract model for ARM Cortex-M, the most popular architecture class for IoT devices, which can be used as a template for building abstract models for other MCU architectures (see §6 for more details). The model generalizes peripheral registers into four types and provides the access patterns and handling strategy for each type (i.e., how an emulator should identify each type of registers and handle each access to a peripheral register based on its type). This access pattern-based register type identification and type-based register access handling is generically applicable to all peripherals on Cortex-M. Therefore, emulators can perform them without requiring any knowledge about specific peripherals (e.g., what kind of peripheral does a register belong to) or knowledge about peripheral internal designs. The model abstracts peripheral interrupt firing as a special input channel and allows customizable interrupt firing strategies. Also included in the abstract model are the locations of the basic memory segments, such as the RAM, the flash, the mapped register region, which remain the same for devices using the same MCU architecture and are specified in MCU documentations[1]. On ARM Cortex-M MCUs, peripheral registers are mapped to `0x40000000`–`0x5fffffff` memory segment as required by the architecture design [41] (i.e., this is an architectural requirement that all hardware and software using this architecture need to follow). P²IM considers each memory word in this segment a potential memory-mapped register.

3.1 Register Category, Access Patterns and Handling

Control Registers: Control registers of peripherals, or `CR`, are used mostly by firmware to control or configure peripherals. For example, firmware sets the corresponding bits in USART's `CR` to enable the transmitter or interrupts or to set the baud rate.

`CR` **Access pattern**: We observed a read-modify-write (RMW) pattern unique to `CR`, whereby firmware first reads a `CR`, then modifies the configuration parameters in it, and finally writes the value back to the register. Firmware follows the RMW pattern when accessing `CR` because firmware can only write at word/register granularity and the RMW pattern avoids inadvertently changing other parameters co-exist in the same register. P²IM uses the RMW pattern to identify `CR`. In some rare case (e.g., a `CR` contains only one configuration parameter), firmware may write directly to it without following the RMW pattern. In this case P²IM, due to the write-on-first-access pattern of `DR` defined below, P²IM can mis-categorize such a `CR` into `DR`. However, in most cases, this mis-categorized register is never read afterward (i.e., for one-time peripheral configuration). Therefore, this kind of register mis-categorization does not impact firmware execution or needs correction.

`CR` **Access handling**: Once a peripheral is configured by the firmware, the peripheral operates accordingly and rarely changes value of `CR` (i.e., peripheral configuration). Therefore, P²IM models each `CR` as a *non-volatile* memory word. When firmware reads a `CR`, the emulator returns the value previously written to the `CR`. If a `CR` is read without being explicitly written before, which can happen after a hardware reset, the emulator simply returns zero, which is the default value for `CR` in most cases.

Status Registers: A status register, or `SR`, is a set of flags (i.e., each flag may contain 1 or more bits) that indicate the internal states of a peripheral. During runtime, peripherals update their `SR` as their status change (peripherals can also use interrupts to notify firmware of status changes, which is discussed in §3.2). Before performing certain peripheral I/O operations, firmware polls the corresponding `SR` bits to make sure the peripheral is ready. For example, firmware reads data from USART only when the data-reception flag in a `SR` is set, indicating some data has been received. Otherwise, firmware simply waits. In many cases, if the necessary `SR` flags are not set, firmware ceases to boot or stalls infinitely (e.g., the system-clock-ready flag in a clock manager peripheral). On

[1] The flash region, where firmware is loaded, may vary on some devices. This information is available in device datasheet.

the other hand, setting the wrong SR bits can cause firmware to crash. For example, certain SR bits being set means fatal peripheral errors, which can switch firmware into recovery or debug mode that requires external intervention. Therefore, properly handling firmware access to SR is critical for achieving P^2IE and undisrupted execution and testing of firmware.

SR **Access pattern**: These registers are used by firmware to check peripheral states. P^2IM categorizes a newly discovered register as SR if the first access to the register is an unconditional read and the read value is later evaluated in a condition. For some SR, the first access on them can be a write, for example, when firmware acknowledging a peripheral error. In this case, P^2IM could initially mis-categorize the SR into DR due to the second DR access pattern defined below. However, P^2IM automatically corrects such mistakes later by leveraging our observation that firmware often reads SR continuously (i.e., to poll peripheral state under synchronous I/O) but not other types of registers. Therefore, when firmware continuously polls on a previously categorized DR, P^2IM adjusts its category to SR and locks its type. We call it polling pattern of SR.

SR **Access handling**: Since P^2IM does not model peripherals themselves and is oblivious to their internal designs, its handling of SR is not based on knowing the semantics of the flags or the registers. Instead, the SR handling aims to dynamically infer an acceptable register value at each SR read so that firmware can continue executing (i.e., the value can pass the firmware's internal checks and lead to subsequent peripheral I/O operations guarded by this SR). P^2IM uses a technique called *explorative execution* to automatically infer acceptable SR values during runtime. This technique belongs to the firmware-specific part of P^2IM (i.e., the model instantiation part), which is discussed in §4. On the other hand, handling SR write is much simpler and the same for all firmware. P^2IM treats SR writes as no-ops, which are ignored by the emulator. This is because SR are volatile and values written by firmware only matter to peripherals internally and are not read back by firmware. In other words, SR write is one-way and the value is transient and does not affect firmware execution. P^2IE can be achieved without handling SR writes.

Data Registers: Data registers, or DR, are the main channel through which raw data flows from peripherals to firmware. Oftentimes, data read by firmware through DR originates from off-chip peripherals (e.g., Zigbee radio) or a remotely connected device (e.g., a supervisory computer in SCADA system). For example, SPI peripheral holds the data it received from an off-chip peripheral (e.g., Zigbee radio) in its DR, which is then read by firmware as input. Data also flows in the opposite direction. Firmware writes output data in the DR and then SPI sends it to an off-chip peripheral.

DR **Access pattern**: Firmware only reads a DR (i.e., taking raw input from a peripheral) after confirming that the peripheral is in a ready state by checking the corresponding SR. Based on this unique access pattern of DR, P^2IM categorizes a newly discovered register as DR if reading the register is preceded by an SR read and conditional on a flag in the SR. Sometimes firmware writes to DR directly without checking any SR. P^2IM uses this write-on-first-access as another access pattern for identifying DR.

DR **Access handling**: DR of all peripherals collectively dominate the inputs to firmware, and therefore, they are ideal ***fuzzing interfaces***. Our framework uses modeled DR to feed fuzzing and testing inputs during runtime. These inputs are generated by a drop-in fuzzer, which may or may not be aware of firmware/peripheral specifics (our current prototype uses unmodified AFL). Upon each DR read, the emulator returns the next word from the fuzzing input as the register value. For other types of dynamic analysis, the input source can be replaced with, for example, previously recorded inputs (for bug/execution reproduction) or specially crafted inputs (for taint analysis). Similar to SR writes, P^2IM ignores DR writes for the same reason that they do not affect firmware execution.

Control-Status Register: A control-status register, or C&SR, is a hybrid register whose bits are split between two purposes: control/configuration bits (same to CR) and status bits (same to SR). Although hybrid registers allow for higher utilization of register bits, they greatly complicate both peripheral hardware and firmware designs. Therefore, they are not commonly used in modern MCU devices, which have abundant memory address space for mapping peripheral registers. In practice, we only observed some rare use of C&SR. We have not seen other types of hybrid registers, such as control-data or status-data registers, which are theoretically possible but impractical.

C&SR **Access pattern**: CR bits of C&SR are modified in the RMW pattern during the peripheral configuration phase. SR bits of C&SR are accessed during the peripheral operation phase. The configuration phase proceeds the operation phase. They do not overlap. As a result, P^2IM often categorizes C&SR as CR in the first place. However, such inaccurately categorized registers are corrected later when P^2IM observes the SR access pattern on them.

C&SR **Access handling**: For each C&SR access, firmware operates on either the CR bits or the SR bits, but not both because they are used at different stages during firmware execution. Since handling SR bit access requires firmware-specific information (similar to handling SR register access), C&SR handling is not covered by the abstract model but by the instantiated model, which is discussed in §4.

Remarks: Although the register access patterns and the type identification method are purely empirical, we find that in practice they work fairly reliably and accurately across a wide range of peripheral devices (see §5 for the evaluation results). We attribute this practical and promising results to two factors: (1) the register types we defined are generically applicable to all peripherals; (2) the type-based access patterns were observed and generalized from a variety of real MCU devices; (3) trade-offs are carefully made when designing the type-based access patterns. Specifically, we observed the

write-on-first-access pattern not only on DR, but also on CR and SR in some occasions. We still use this pattern to identify DR despite that certain CR and SR might be mis-categorized. This trade-off is made and justified by the following considerations. First, this pattern is most commonly seen on DR. By using this pattern for detecting DR, we can achieve the best overall register categorization accuracy. Second, a SR mis-categorized by this pattern is often corrected later on by the polling pattern unique to SR (i.e., this mis-categorization is temporary). Third, a CR mis-categorized by this pattern (e.g., a CR contains only one configuration parameter) does not impact firmware execution/testing or needs correction because firmware generally does not read or take input from CR.

3.2 Interrupt Firing

Apart from the register categories and handling, the abstract model also defines how emulators should fire necessary interrupts on behalf of peripherals in order to satisfy P^2IE and support continuous firmware execution or testing.

In essence, interrupts are a special type of inputs to firmware. They notify firmware of certain hardware events and trigger the corresponding interrupt service routines (ISR), which are interrupt handlers implemented by peripheral drivers in firmware. For instance, an interrupt may signal the firmware that input data is ready. Then the corresponding ISR is invoked and reads the input data from a DR.

A processor emulator, such as QEMU, often includes a virtual interrupt controller, which could dispatch fired interrupts to software. However, since these emulators do not emulate MCU peripherals, no peripheral interrupt is fired when using them to run a firmware, despite that the firmware may crash or stall for other reasons before it gets ready for servicing interrupts.

P^2IM abstractively models interrupts as a sequence of timing-based inputs, with each input corresponding to an enabled interrupt. When such an input comes in, the emulator generates and dispatches the matching interrupt to the firmware. The emulator detects what interrupts are enabled by the firmware during runtime (discussed in §4.3). P^2IM allows both the sequence and the timing of interrupts to be customized based on different fuzzing strategies (e.g., purely random generation, mutation from crafted seeds, etc.). Our current prototype uses a simple interrupt firing strategy: enabled interrupts are fired in a round-robin fashion at a fixed interval (e.g., after every 1,000 basic blocks executed). The interval is defined using the number of executed basic blocks, rather than absolute time (e.g., clock ticks). This basic block-based interval definition supports arbitrary timings to be specified for emulators to fire interrupts. More importantly, using basic block counts to measure interrupt intervals allows for deterministic replay of interrupt sequences, and therefore, yields reproducible fuzzing/testing results. This reducibility is required for fuzzing, but usually hard to achieve when using existing fuzzers with asynchronous interrupts enabled.

We note that defining more advanced interrupt firing strategies is the job of drop-in fuzzer or fuzzer operators and is out of the scope for P^2IM. Although the current interrupt firing strategy is simple, it already leads to a very high firmware code coverage as shown in the evaluation section.

3.3 Infeasible Peripheral Inputs & False Positives

Under the current abstract model definition, P^2IM can trigger code paths in firmware that are infeasible on real devices. This is because hardware peripherals may only generate certain inputs and fire interrupt at certain patterns, whereas P^2IM allows fuzzers to generate random peripheral inputs or adopt arbitrary interrupt timing. Such infeasible inputs and code paths may cause false positives during fuzzing (i.e., a crash/hang is caused by an infeasible input/path, rather than by a firmware bug). However, as a generic firmware testing framework, P^2IM does not prune potentially infeasible inputs or code paths. Instead, P^2IM leaves the task of input pruning, which is part of the input generation process, to the testing tools running on top of P^2IM (e.g., a fuzzer). This design decision is made for two reasons. First, input generation and input quality control are among the core tasks of fuzzers and other dynamic testing tools. P^2IM is designed to support these tools and not in the position to interfere with these tasks. Second, as observed in [47], peripherals such as Wi-Fi radio are vulnerable to remote attacks (e.g., an attacker sending malformed network packet), and once compromised, can generate unexpected input or interrupt timing. Therefore, testing firmware with unexpected/infeasible peripheral inputs might be desirable in some cases.

Nevertheless, we did not see in our extensive experiments any crashes/hangs that were caused by the infeasible inputs or code paths introduced by P^2IM, despite that no input pruning was performed. All crashes/hangs detected on P^2IM were reproducible on real hardware (i.e., they were caused by true bugs in tested firmware), except for two false hangs that were caused by P^2IM. We analyze these two cases and the limitations of P^2IM that caused them in §5.3.

4 Automatic Model Instantiation

As illustrated in Figure 3, the second step of P^2IM is the instantiation of an abstract model defined in the first step, producing a full model for a given firmware. The instantiated model guides the emulator to identify and handle I/O operations through the process-peripheral interfaces. During the instantiation step, the firmware-specific information needed for providing P^2IE is added to an abstract model.

The model instantiation process is fully automatic and uses the explorative execution technique. The instantiation is on-demand and interleaved with the firmware fuzzing/testing process. The fuzzing process invokes the model instantiation process when it encounters unmodeled or unhandled

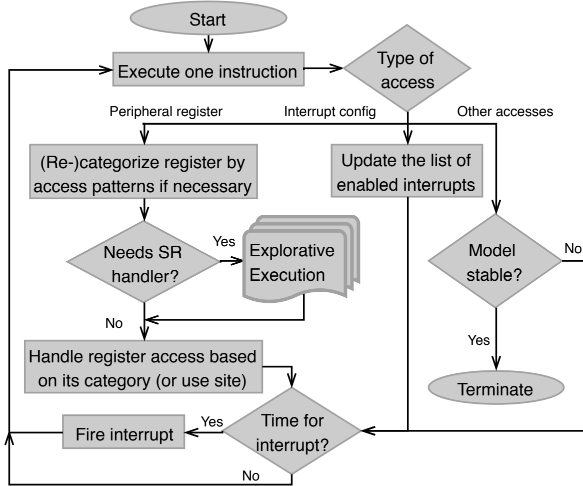


Figure 4: Model instantiation workflow

peripheral access. The model instantiation process terminates and the fuzzing resumes when the model becomes stable and no new information (e.g., newly identified registers) is added to the model for a while. Note that this switch is not triggered by the model reaching a certain level of precision. We call each invocation of the model instantiation process "one round of model instantiation". Multiple rounds of model instantiation can happen at different points throughout a firmware fuzzing/testing process. The model instantiation process is deterministic and repeatable. An instantiated model can be reused for the same firmware. The process relies on a customized QEMU that emulates the Cortex-M instruction set and a generic interrupt controller but not MCU peripherals.

Specifically, an instantiated model contains the following automatically inferred firmware-specific information, which concretizes the abstract model: (1) identified memory-mapped registers, their memory locations, and types; (2) the access handling strategies for each type of registers, or each use site when needed; (3) the enabled interrupts and the firing strategy. An instantiated model does not contain any information about peripheral configurations or internals.

Figure 4 shows the high-level workflow of the model instantiation process. It executes the firmware on a generic processor emulator which does not emulate any peripherals. It continuously instantiates the model when firmware accesses the processor-peripheral interfaces. Upon a register access, it (re-)categorizes the register if necessary as described in §4.1. Then it handles the register access as described in §4.2. Specially, for an SR read, it checks whether a handler exists. If not, it performs the explorative execution to automatically generate a handler (§4.2). It also monitors interrupt configurations by the firmware and fires interrupts when needed (§4.3)

4.1 Register Identification

The goal of register identification is to detect the memory-mapped registers exposed by peripherals and determine their types according to our category and access pattern definition (§3.1). It identifies and categorizes all registers that are accessed by the firmware as it runs on P^{2}IM. During the model instantiation process, P^{2}IM monitors firmware's access to the memory segment reserved for peripheral registers (e.g., `0x40000000`–`0x5fffffff` on ARM Cortex-M MCUs [41], as captured in the abstract model). P^{2}IM considers each accessed memory word in this segment a memory-mapped register.

Although detecting such registers is straightforward, determining their types is fairly challenging because P^{2}IM or the emulator does not have any knowledge about the semantics of the registers or the peripherals that the registers belong to. Overcoming this challenge, P^{2}IM determines the type of a newly identified register based on the per-type register access patterns that we empirically observed and generalized from a large set of MCU peripherals (§3.1).

Peripheral Association: In addition to identifying peripheral registers and their types, P^{2}IM also groups them based on if they belong to the same peripheral. This grouping is needed for accurately handling certain `SR` accesses (discussed in §4.2). It only considers peripheral association and is unaware of peripheral types or characteristics. P^{2}IM identifies the groups based on the spatial adjacency and alignment of registers' memory addresses.

4.2 Register Access Handling & Explorative Execution

P^{2}IM provides strategies for type-based register access handling, which instructs the emulator, upon register access, what actions it should take, including return what value to the firmware or what internal state to update. The strategies for handling `DR` and `CR` accesses are straightforward and uniform across firmware. They are part of the abstract model defined in §3.1.

On the other hand, the strategies for handling `SR`, including the `SR` bits in `C&SR`, are much more complicated. The strategies may vary across different `SR` as well as different use sites of the same `SR`. For example, the most significant bit in two different `SR`, SR_1 and SR_2, have completely different meanings. At one point of firmware execution, setting the bit in SR_1 is needed for the firmware execution to continue without stalling but setting it in SR_2 crashes the firmware. At different points of firmware execution, setting the bit in SR_1 causes the opposite. Therefore, strategies for handling `SR` accesses need to consider firmware specifics, individual registers, and their access contexts. As part of the model instantiation step, P^{2}IM automatically derives `SR` handling strategies using a technique called *explorative execution*, which is the focus of this subsection.

The high-level idea of the explorative execution is as follows. When the firmware execution encounters a new SR access site, P^{2}IM pauses and snapshots the execution and starts the explorative execution. By spawning multiple parallel worker threads, the explorative execution concurrently searches for the best value for the SR, resumes the original firmware execution, and returns the SR value to the firmware. The key challenges addressed by our design of this technique include: constructing a tractable search space of candidate SR values; determining the scope of the explorative execution (or the termination condition for the workers); defining what qualifies the best SR value; reducing the frequency of explorative executions. We describe our solutions to these challenges below.

Search Space Construction: The search space could be intuitively constructed by including all possible values for an SR. This interprets to 2^{32} candidate values on a 32-bit MCU and in turn requires the explorative execution to spawn 2^{32} worker threads to test the candidate values, which obviously is infeasible. Instead, we construct a much smaller and tractable search space by taking advantage of the fact that bits/flags in SR are usually independent and only a single flag is checked at a time. Our search space contains only 32+1 candidate values, each with a single bit set in an SR plus a zero (all bits clear). The explorative execution spawns one worker thread to test each candidate value. In each thread, the candidate value is returned to the firmware as the value of the SR. All worker threads execute in parallel. P^{2}IM monitors their progress and picks a winner (i.e., the thread with best candidate value) at the end of the explorative execution.

Termination of Explorative Workers: When the explorative execution should terminate a worker thread is another design question. If too early, the worker thread may have not reached the use of the SR that is critical to firmware execution. If too late, the explorative execution becomes too long and can cause significant delay or even halt the firmware execution (e.g., due to encountering another SR read whose access handling strategy has not been derived yet). We experimented several termination conditions and different life spans of work threads. We found one that works well in practice and keeps the runtime overhead low. It terminates a worker thread when it is about to return to the next level callee (i.e., when the current call stack frame, where the SR read happened, is popped). The rationale is that firmware usually reads an SR in the same function where it decides if further I/O operations can be performed based on the SR value. Therefore, having the explorative execution continue beyond function boundaries does not yield additional benefits for finding the best SR value, despite that it significantly complicates the thread monitoring mechanism and slows down P^{2}IM. It is worth noting that many worker threads exit before they reach the termination point because the assigned SR values are unacceptable to the firmware.

Qualified Workers and SR Values: After all worker threads terminate, P^{2}IM determines which threads or candidate SR values qualify for potentially advancing firmware execution. It then picks the best among the qualified values to return to the original firmware execution, which concludes the explorative execution. The qualification criteria are: (1) the thread did not crash or stall; (2) if all threads crashed or stalled, choose those caused by other factors than the current SR (i.e., the crash/halt site is not dependent on the SR value). Among the qualified worker threads and the candidate SR values they represent, P^{2}IM selects the best based on the number of DR accesses, guarded by the SR, that were observed during thread execution. When multiple equally good SR values are found, P^{2}IM randomly picks one as the best value (and records the choice to make the model instantiation process deterministic and repeatable). The design of the worker qualification and selection aligns with the definition of P^{2}IE: input values that are acceptable to firmware and unlock meaningful operations are used in place of inputs from real peripherals to achieve P^{2}IE and sufficient for supporting firmware fuzzing/testing.

Minimizing Explorative Executions via SR Grouping: The design of the explorative execution discussed so far treats individual SR accesses independently. If implemented as is, this design can cause frequent explorative executions (upon every SR read during firmware execution) and thus high overhead. But on the other hand, access handling strategies need to be derived for every use of every SR as explained earlier. We address these two conflicting needs by optimizing our design via SR grouping. The idea is that an access handling strategy derived for one SR at one location, though not universally applicable to all SR, can be reused for the same SR accessed in similar locations. Specifically, we group SR accesses based on their context, defined by a four-tuple $(r, cs, bbl, conf)$, where r is the SR; cs is the signature of the call stack at the time of the SR access; bbl is the ID of the basic block in which the SR read occurred; $conf$ is the peripheral configuration hash trivially generated from CR values at the time of the SR access, which does not contain any semantic information for peripheral configurations (such as whether the receiver is on or off). The configuration hash is included because different CR values cause the firmware to check SR differently. For example, firmware only checks the data-reception flag of USART when the receiver is enabled via CR. With SR grouping, similar SR accesses can reuse the same handling strategy, which increasingly reduces the frequency of explorative executions as P^{2}IM instantiates the model.

4.3 Interrupt Identification

Another task that P^{2}IM performs during the model instantiation step is collecting the firmware-specific information about interrupts. MCU architectures (e.g., Cortex-M) support hundreds of different interrupts. But a particular device or its firmware may only use a small subset of supported interrupts. Moreover, during runtime, firmware sometimes dynamically

disables and re-enables interrupts as needed. If an emulator fires an unused or disabled interrupt, firmware execution can stall or crash because firmware commonly uses a simple dead loop as the default handler for unused interrupts.

P^2IM maintains a list of currently enabled interrupts during firmware execution. It taps into the virtual interrupt controller of QEMU (Nested Vectored Interrupt Controller, NVIC), which the firmware configures to enabled/disable interrupts. Drawing from the list of enabled interrupts, the interrupt firing strategy defined as part of the abstract model (§3.2) decides when to fire what interrupts.

4.4 P^2IM Implementation

We implemented our framework using QEMU as the base processor emulator (without any peripheral emulation capability). Our implementation includes 2,202 lines of C code added to QEMU (mostly for dynamic firmware execution instrumentation), 173 lines of C code for fuzzer integration, and 1,199 lines of Python code for the explorative execution part of P^2IM. We use AFL as the drop-in fuzzer in our current prototype, which has no built-in support or awareness of MCU firmware.

We implemented register categorization, peripheral identification, type-based register access handling, and SR read grouping logic inside two QEMU functions, namely `unassigned_mem_read` and `unassigned_mem_write`, where accesses to memory-mapped peripheral registers are directed to. For fast prototyping, we implemented the complex logic of explorative execution using Python. But the worker threads still run natively on QEMU. The interrupt identification and firing logic are implemented based on the QEMU's virtual interrupt controller (NVIC). The logic monitors firmware's accesses to `NVIC_ISERx` and `NVIC_ICERx` registers to detect enabled interrupts. It fires interrupts via the `armv7m_nvic_set_pending` interface exposed by NVIC.

AFL's emulation mode (used for fuzzing un-instrumented binaries) only supports user-mode emulation, which is incompatible with firmware emulation [12]. TriforceAFL [37] builds a bridge for AFL to be connected to the full system emulation mode of QEMU. We used TriforceAFL's code when implementing the fuzzer integration part of our framework, which allows fuzzers to be dropped in without modifications. During runtime, the fuzzer integration code channels inputs generated by the fuzzer to firmware execution through `DR` accesses. It collects code coverage information via the QEMU instrumentation and returns the information to the fuzzer.

5 Evaluation & Fuzzing Results

We evaluated our framework from three different angles: (1) whether it satisfies P^2IE when executing firmware for different MCU with different OSes; (2) how its runtime performance is in practice; (3) whether it can perform fuzz-testing on real firmware in a fully emulated fashion (i.e., no hardware dependence) and find previously unknown bugs.

Table 1: Selected peripherals & functional operations

Peripheral	Functional Operations
SPI	Receive a byte Transmit a byte
USART	Receive a byte Transmit a byte
I2C	Read a byte from a slave Write a byte to a slave
GPIO	Read status of a pin Set/Clear a pin Execute callback after pin interrupt
ADC	Read an analog-to-digital conversion
DAC	Write a value for digital-to-analog conversion
TIMER	Execute callback after interrupt Read counter value
PWM	Configure PWM as an autonomous peripheral

To that end, we performed functional unit tests based on commonly used MCU peripherals and different MCU OSes (§5.1). We also conducted an end-to-end test on real firmware (§5.2). Finally, using our framework, we performed fuzzing on real firmware, found bugs, and gained interesting insights (§5.4). We conducted all experiments on a moderate-spec computer with a dual-core Intel® Core™ i5-7260U CPU @ 2.20GHz, 8 GB of RAM, and Ubuntu 16.04.

5.1 Unit Tests on MCU Peripherals & OSes

We designed and performed this experiment to verify if our framework can indeed provide P^2IE when fuzzing firmware that: (1) access a range of peripherals (i.e., P^2IM provides generic peripheral support), (2) are designed for different MCU SoCs (i.e., P^2IM is applicable to a broad class of MCU), and (3) use different OS/system libraries (i.e., P^2IM is OS agnostic). For this purpose, we collected a set of example firmware as unit test cases for this experiment.

Experiment Setup: We identified the 8 most popular MCU peripherals—implemented as on-chip peripherals—by analyzing the entire MCU product line (1686 MCU parts) offered by Microchip Technology, a top global MCU vendor. We selected these peripherals (the left column in Table 1) for our unit tests. We also selected 3 widely used MCU OS/system libraries, (NuttX, RIOT, and Arduino) and 3 target MCU SoCs (STM32 F103RB, NXP MK64FN1M0VLL12, and Atmel SAM3X8E). We selected these SoCs because they are part of the reference designs provided by major MCU vendors, and had been integrated into automotive/marine [2], consumer [17] and healthcare [11] products.

We collected 70 different example firmware or test cases, each representing a unique and feasible combination of a peripheral, an OS, and a SoC. After booting, these firmware simply perform the basic peripheral operations defined in

Table 2: Peripherals and registers accessed during unit tests

	Peripherals accessed			Registers accessed		
Peripheral	**Max.**	**Min.**	**Avg.**	**Max.**	**Min.**	**Avg.**
I2C	15	5	9.0	54	18	35.6
ADC	14	6	8.8	68	30	46.0
PWM	14	7	10.2	62	25	43.2
TIMER	14	7	9.7	47	26	38.0
GPIO	13	3	7.7	57	9	34.3
SPI	13	6	8.3	66	19	36.8
USART	13	4	7.5	53	15	30.0
DAC	11	8	9.5	60	35	47.5

Table 1. We made sure these firmware run smoothly on their target SoC. We then run them using our framework and collect the statistics and results, including the accuracy of the instantiated model and end results.

Experiment Results: We collected the statistics on peripherals and registers accessed during the tests, as shown in Table 2. This shows that a single peripheral operation often incurs multiple accesses to related or dependent peripherals of different kinds. For example, the minimum number of peripherals accessed for during the `GPIO` test is 3 and the maximum number is 15 for `I2C`. Additionally, multiple register accesses are associated with a single peripheral operation. For instance, the minimum number of involved registers for a `GPIO` operation is 9 and the maximum number is 68 for `ADC`. These statistics show that even a simple peripheral operation can involve a complex chain of other peripherals and many registers, highlighting the value of P^{2}IM and the need for automatic modeling and handling of MCU peripherals.

We also measured the accuracy of register identification and categorization. We first manually extracted the ground truth from the MCU datasheets and then compared the register categorization output from our system with the ground truth. Figure 5 c) shows the result aggregated by peripherals, ranging from 76% to 92% (i.e., 24% to 8% identified registers are mis-categorized). There are no particular peripherals on which P^{2}IM performs much better or worse than others. This suggests that P^{2}IM's accuracy does not vary much across different types of peripherals. It also echos that P^{2}IM is oblivious to peripheral types or internals. We discuss the reasons of register mis-categorizations and their impact on firmware execution in §5.3.

The unit test result is that 79% (55) of the tests passed (i.e., P^{2}IE was satisfied) while 21% (15) failed. For a test to qualify as pass, the firmware under test needs to properly boot, configure the peripherals, and conduct the functionality, without any crash, stall, or operation skipping. The pass rate of 79% may not seem very high at first glance. But considering it represents an improvement from 0% (i.e., no previous work can generically and automatically model MCU peripheral I/O), we argue the result is in fact significant. The per-MCU and per-peripheral breakdowns are shown in Figures 5 a) and b). The former shows that P^{2}IM performs equally well across different MCU SoC and OS combinations, suggesting it is MCU- and OS-agnostic. The latter reveals that P^{2}IM encountered failures on `USART`, `TIMER`, `I2C`, and `GPIO` but not the other peripherals.

We found 2 general causes for failed tests. First, register mis-categorizations can happen when peripheral drivers fail to follow the correct/common register access patterns. In our experiments, we observed the majority of failures on `GPIO` and `I2C` peripherals are due to this reason. Second, some peripherals multiplex individual interrupts, which can cause P^{2}IM to fire incorrect interrupts.

Overall, this experiment shows that P^{2}IM works reasonably well on a large set of example firmware (using real drivers and system libraries and no accommodation to P^{2}IM). It allows most of the firmware to execute, without any crash, stall, or operation skipping, on an emulator that does not support MCU peripherals. Moreover, P^{2}IM is shown to be agnostic to firmware's target MCU and OS.

5.2 End-to-end Tests against Real Firmware

This experiment examines the performance of our framework when running and testing real MCU firmware of different kinds. These firmware are much bigger and more complex than the unit test firmware used in the previous experiment, although the unit tests are larger in quantity and more diverse in terms of the used peripherals and target MCU and OSes. This and the previous experiment together show P^{2}IM's ability of handling diverse and complex firmware.

Experiment Setup: We selected 10 firmware of real MCU-based devices used for different purposes, ranging from drones to industrial control systems. They are full-fledged firmware and contain all the common firmware components, including the kernel (e.g., scheduler, interrupt handler, system libraries), drivers, console, application logic, etc. They collectively cover 4 MCU models (from 3 top MCU vendors by revenue [9]), 4 OSes and a diverse set of peripherals (Table 7). Moreover, the underlying SoC used in these firmware are often used in other embedded or IoT devices. We evaluated both the model instantiation mechanism (in the current section) and the fuzzing performance (§5.4) on the 10 selected firmware. The details about the 10 firmware are presented in Table 7 in Appendix A. A brief description for each firmware is as follows:

Self-balancing Robot: This is the motion controller firmware in a robot architecturally similar to commercial personal transporters (e.g., Segway PT). Even basic vulnerabilities in such firmware, such as integer overflows, can lead to disastrous consequences or life-threatening accidents.

PLC (Programmable Logic Controller): PLC is a rugged embedded device for controlling critical processes in industrial environments (e.g., assembly lines). This selected

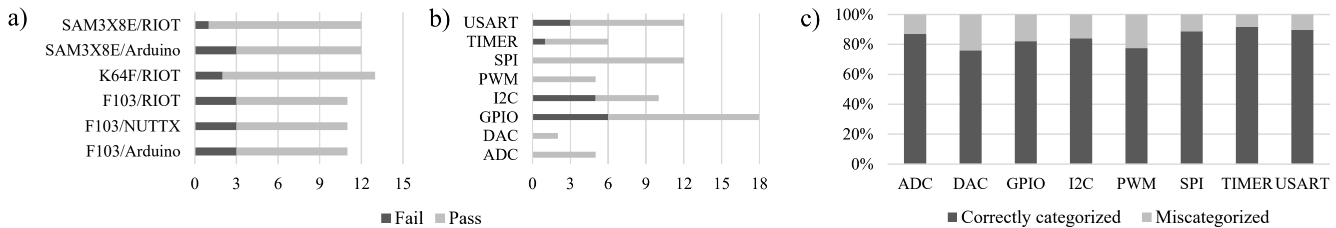


Figure 5: Unit test results aggregated by MCU SoC/OS (a) and Peripheral (b). Accuracy of register categorization (c)

firmware is part of a sterilizer machine and manages a PLC's communication with remote SCADA (Supervisory control and data acquisition) systems via Modbus, an industrial communication protocol [10]. Vulnerabilities in PLC firmware are often critical as demonstrated by the Stuxnet attack [50].

Gateway: This firmware is for a gateway device that uses the Firmata [6] protocol to communicate with its host computer, allowing the host computer to access and configure MCU peripherals dynamically, such as sensors and actuators. Security vulnerabilities in such firmware can be exploited to remotely hijack/abuse embedded devices.

Drone: This firmware drives the MCU-based autopilot controller in a quad-copter similar to the Pluto Drone [35]. It controls multiple sensors, radios, motors, etc. and implements a suite of control algorithms, such as PID (proportional, integral and derivative). Vulnerabilities in drone firmware can be exploited to manipulate drones and cause physical damage.

CNC: This firmware is a Cortex-M port of the widely used Grbl milling controller [7]. Grbl has been used in many commercial and open-source 3D printers, laser cutters, hole drillers, etc. This firmware includes a G-code interpreter, the linear/circular interpolation algorithms, and the stepper-motor control routines. Vulnerabilities in the G-code interpreter or control routines can lead to physical injuries of machine operators or destruction of the milling equipment.

Reflow Oven: This firmware is for a commercial-grade reflow oven [14] controller used for assembly of printed circuit boards (PCB). This controller implements push-buttons and LCD as user interface, thermocouple input, acoustic alarm, and dual output for the heating element and oven fan. The temperature profile of controller is based on the multi-ramp [24] PID control loop. Vulnerabilities in this firmware can compromise the industrial processes and the quality of PCB assembly.

Console: This firmware implements all the standard utilities of the RIOT OS and exposes the shell through a serial console. The shell of RIOT implements a small but powerful interface to execute user-defined callbacks and other system utilities for control and diagnostic purposes. Vulnerabilities in the shell can compromise internal data structures of the OS and even expose a device to remote code execution.

Steering Control: This firmware implements the algorithm of a steer-by-wire [20] controller deployed in a lab-grade self-driving vehicle. It takes commands from the main on-board computer and translates them to electrical signals to control servomotors. Bugs in this or similar type of devices are the causes of multiple documented deadly car accidents, plaints and recalls from major automotive companies [27].

Soldering Iron: This firmware is an open-source version of the popular "TS100" soldering Iron. It implements an LCD and several push buttons for adjusting temperature and other parameters. Internally, it runs a PID control algorithm and an acceleration sensing routine for auto-power off. Vulnerabilities in these devices can lead to overheating of the soldering iron, which can cause damages to the objects being soldered or injuries to the operators.

Heat Press: This firmware corresponds to an industrial heat press [8] used in a textile sublimation production line. The firmware implements recipe manager for controlling the temperature, time and pressure of the sublimation process. The system features a touch screen and a remote industrial I/O channel using the Modbus protocol. Vulnerabilities in this type of systems can lead to unintended operations, remote hijacking, and damage of industrial facilities.

Experiment Results: Our framework achieved similar or even better results on real firmware than on the unit tests. As shown in Table 3, the register categorization accuracy (Acc.%) is even higher than measured in the unit tests, despite that the real firmware are more complex and access more registers and peripherals. Our manual verification attributes this better result to the fact that these firmware follow the register access patterns more strictly than the sample drivers in the unit tests. We explain the reasons and impacts of register mis-categorizations in §5.3. The last column of Table 3 shows the total time (in seconds) spent on model instantiation for each firmware. The time consumed (10 minutes in the worse case) is acceptable given that a typical fuzzing session often lasts for days or longer.

Figure 6 shows the progress of the model instantiation for each firmware. As explained in §4, P^{2}IM launches a new round of model instantiation when it encounters an unmodeled or unhandled peripheral interface access. For most firmware, P^{2}IM instantiated the models within 3 rounds. Note that the last few rounds of model instantiation for PLC, Drone, HeatPress and Soldering Iron formed new SR read groups, which are not shown in the figure for simplicity. Gateway incurs 25 rounds model instantiation because it initializes peripherals on-demand (i.e., additional model instantiation

Table 3: Model instantiation statistics on 10 real firmware

Firmware	Peri.	Regs.	Acc. (%)	SR group	Int. line	Time (s)
Robot	7	43	100.0	16	1	131.2
PLC	5	21	100.0	5	2	6.8
Gateway	14	101	93.4	14	5	612.4
Drone	11	68	100.0	20	2	315.7
CNC	12	81	91.5	5	2	48.3
Reflow O.	6	32	95.8	1	2	4.4
Console	11	43	88.5	9	1	28.0
Steering C.	11	79	69.6	14	3	96.2
Soldering I.	13	84	90.7	33	9	512.0
Heat Press	4	76	84.0	25	2	59.4

is needed when a new peripheral is initialized after the model has stabilized). On the 10 tested firmware, P^{2}IM triggers a round of model instantiation every 2,579 seconds, on average, until all peripheral interfaces are modeled.

In most cases, P^{2}IM automatically derived proper models to support firmware execution with P^{2}IE satisfied (i.e., no crash, stall, or operation skipping). We will present the two cases that break P^{2}IE in §5.3. We then used these models to perform fuzzing on all the firmware (§5.4).

This end-to-end experiment on real firmware and the unit tests firmware together confirm that our framework achieves its goal: enabling hardware-independent and scalable firmware testing/fuzzing via P^{2}IM. Moreover, the overhead and inaccuracy are low enough for our framework to be used in practice.

5.3 Register Mis-categorizations & False Crashes/Hangs

In this section, we discuss the two types of false positives, namely register mis-categorizations and false crashes/hangs, that our mechanism may cause.

Register Mis-categorizations: We manually examined all the registers mis-categorized by P^{2}IM while being tested against the 10 real firmware. We itemized their potential impacts on firmware execution and present number of mis-categorized registers per impact in Table 4. For each impact, we give a representative example of the mis-categorized register and explain why the register is mis-categorized.

When calculating the register categorization accuracy, we only consider registers that have been read at least once during firmware execution (i.e., registers never read are not counted because they do not affect the firmware execution). In Table 4, the last column shows the total number of registers that have been read by the firmware. The middle two columns show the number of registers mis-categorized by P^{2}IM for each firmware. The mis-categorized registers are grouped in two types based on their negative impact to firmware fuzzing (i.e., slowing down fuzzing or reducing the coverage). The overall accuracy of register categorization (the "Acc." column in Table 3) for each firmware is calculated as follows: $Accuracy = 1 - (TypeI + TypeII)/TotalRegistersRead$.

Table 4: Numbers of mis-categorized registers on 10 real firmware, grouped by their impacts: either slowing down fuzzing (Type I) or limiting coverage (Type II). The last column shows the total number of registers that have been read during the firmware fuzzing process.

Firmware	Mis-cat. Regs (Type I)	Mis-cat. Regs (Type II)	Total Regs Read by Firmware
Robot	0	0	25
PLC	0	0	18
Gateway	4	0	61
Drone	0	0	39
CNC	1	3	47
Reflow O.	0	1	24
Console	0	3	26
Steering C.	6	1	23
Soldering I.	4	1	54
Heat Press	4	0	25
Total # (%)	19 (5.6%)	9 (2.6%)	342 (100%)

Type-I mis-categorizations, mostly `SR` mis-categorized to `DR`, may slow down the fuzzing process but do not stop or break it. The reason is that this type of mis-categorizations causes the fuzzer to guess the `SR` value that is supposed to be quickly generated by P^{2}IM. Nonetheless, AFL can effectively guess the proper value using the coverage information as guidance. Mis-categorized registers of this type are caused by certain peripheral drivers not following the access patterns we defined to categorize registers. We analyzed these cases manually and found that these drivers should have followed the access patterns to avoid potential I/O errors. One exception is that the firmware writes `SR` (to clear potential peripheral errors) before ever reading it (to check the peripheral state), which causes the `SR` to be mis-categorized to `DR` due to the write-on-first-access pattern for identifying `DR`.

Type-II mis-categorizations, mostly `DR` mis-categorized to `CR`, prevent the fuzzer from reaching some code paths that depend on the input from the mis-categorized `DR`. Such mis-categorized `DR` may stall part of the firmware (e.g., one thread of the Soldering Iron firmware) and partially break P^{2}IE. This is because P^{2}IM is unable to channel the fuzzer-generated input into the firmware through the mis-categorized register. The cause of such register mis-categorizations, similar to the previous type, is that drivers fail to follow the common/correct register access patterns to avoid potential I/O errors. One exception is that some GPIO peripherals expose multiple pins via one `DR`. To write data to a pin, a driver has to follow the RMW pattern to avoid overwriting other pins, which causes `DR` to be mis-categorized into `CR`. This is a limitation of our register categorization method.

In summary, only 8.2% registers read by firmware were mis-categorized, and less than one third (2.6% out of 8.2%)

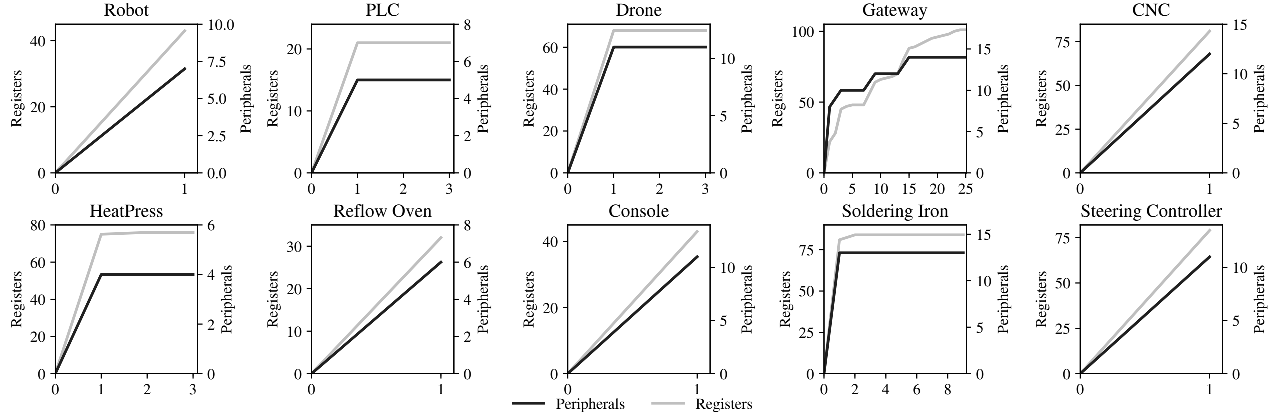


Figure 6: Per-round Progression of Model Instantiation (registers and peripherals covered) on 10 Real Firmware

of the mis-categorized registers negatively impacted P²IE. This is empirically acceptable because, as shown in §5.4, P²IM achieves good fuzzing performance on real firmware despite the (rare) mis-categorized registers. P²IM enables hardware-free and scalable MCU firmware testing. It achieves high code coverage and finds previously unknown bugs, with no false crashes and very few false hangs.

False Crashes/Hangs: In our evaluation, we found that no crashes/hangs are caused by infeasible input or code paths introduced by P²IM (§3.3). This justifies our design decision that leaves the task of input pruning to the fuzzer. However, we found two hangs on the Soldering Iron firmware caused by the limitations of P²IM: one is caused by a `DR` that is mis-categorized as `CR` (discussed above) and the other is due to firmware's usage of DMA, which is very difficult to model automatically and we consider it out of the scope for this paper.

Except for the two hangs, all other crashes/hangs found by P²IM are caused by real bugs inside the firmware. We verified this by running the firmware on real devices with the same inputs that caused crashes/hangs on P²IM, and then confirmed that these inputs also cause crashes/hangs on the real devices.

5.4 Fuzzing Results & Case Studies

In this section, we demonstrate that P²IM, without requiring any MCU hardware or peripherals, is able to fuzz-test real MCU firmware and find previously unknown bugs. In our experiments, we used the unmodified AFL as the drop-in fuzzer for P²IM[2]. We did not use or evaluate other fuzzers because finding or designing a better fuzzer is not the goal of this work. P²IM feeds AFL-generated inputs to firmware execution via `DR` (identified during the model instantiation process) when accessed. P²IM collects and sends the execution coverage to AFL as the fuzzing feedback.

For simplicity, we used randomly generated seed inputs (i.e., no expert knowledge about firmware input is given to the fuzzer). For testing purposes, our framework uses a very basic memory error detector, which is based on the segment tracking heuristics described in [43]. It enforces the least permissions needed by each memory segment: `R+X` for flash, `R+W` for RAM, peripherals, and system control block, and `no-access` for the rest of the memory. As a result, it can only detect a small set of bugs (i.e., memory corruptions that span region boundaries and violate the permissions).

[2]Any existing or future fuzzers can be used as a drop-in component

Table 5: Unique bugs detected in real firmware

Firmware	CWE*	Unique bugs
PLC	704, 129, 787	3
	190, 129, 787	1
	681, 129, 400	1
Gateway	129, 787	1
Heat Press	129, 787	1

* Common Weakness Enumeration (www.mitre.org)

After fuzzing each firmware for 24 hours, our framework found 7 unique and previously unknown bugs in the firmware (Table 5). All of them were later confirmed as exploitable by our manual analysis. Based on the result, we can reasonably anticipate that our framework is likely to find even more bugs in these firmware by using expert-crafted seed input and an advanced MCU memory sanitizer (both are orthogonal to the topic of this paper). We reported all the bugs to the device vendors.

As shown in Table 5, the bug found in Gateway is a combination of an improper validation of array index (CWE-129) and an out-of-bound write (CWE-787). This bug allows a remote attacker to overwrite data objects on the embedded device and cause denial of service or data corruption. The bug found in Heat Press shares a very similar nature with the Gateway bug. The five bugs found in PLC, three similar and two distinct in nature, are combinations of the common programming errors, such as incorrect type cast (CWE-704), integer overflow (CWE-190), incorrect conversion between numeric types (CWE-681), and

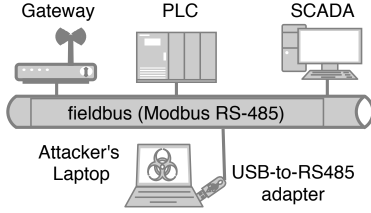


Figure 7: PLC device setup

Table 6: Fuzzing performance on 10 real firmware

Firmware	Speed (run/s)	Basic block coverage w/o P²IM	w/ P²IM	Improv.
Robot	29.5	2.5%	43.1%	17.0x
PLC	32.7	3.5%	26.1%	7.6x
Gateway	17.8	1.7%	45.2%	26.5x
Drone	17.2	8.4%	58.4%	6.9x
CNC	18.0	2.7%	69.5%	26.1x
Reflow O.	24.7	3.6%	39.8%	11.2x
Console	14.6	2.2%	37.8%	17.2x
Steering C.	32.3	0.7%	19.8%	29.5x
Soldering I.	13.5	4.2%	53.2%	12.7x
Heat Press	39.4	1.1%	28.1%	24.8x

uncontrolled resource consumption (CWE-400).

Some of these bugs are more critical than others due to the possibility of arbitrary memory read/write by remote attackers. To demonstrate the real security impact, we developed a proof-of-concept (PoC) for the PLC bugs. As shown in Figure 7, the PLC device is typically attached to a fieldbus through Serial Port. Any malicious devices on the bus, either owned by an adversarial insider or compromised by a remote attacker, can exploit the PLC bugs we found by sending crafted commands over the fieldbus. By the PoC, we modified the internal memory arrays in PLC which contain the critical parameters for PID (Proportional, Integral, Derivative control algorithm). Using this PoC, an attacker can directly influence the PLC-controlled industrial process, causing Stuxnet-like damages.

This fuzzing experiment not only demonstrates our framework's ability to find bugs in real firmware but also shows its relatively high level of code coverage and fuzzing speed. Table 6 shows the fuzzing speed (number of fuzzing runs per second), the basic block coverage without P^{2}IM, the basic block coverage with P^{2}IM, and how much coverage P^{2}IM improves. With P^{2}IM, the code coverage improved 7 to 30 times from the coverage without P^{2}IM, echoing the value of P^{2}IM and the importance of (automatic) peripheral I/O handling.

The much-improved code coverage may still seem low number-wise. We investigated it and found four main causes for it. First, these firmware tend to contain dead code as regular software does (i.e., unused or fractionally used libraries). Second, AFL is a simple grey-box fuzzer that is not good at breaking through complex path conditions (e.g., checksum checks). Such inputs and path conditions are commonly seen in firmware. Since P^{2}IM hosts fuzzer as a drop-in component, we can replace AFL with a more advanced fuzzer such as [49] to overcome this problem. Third, the two false hangs on Soldering Iron firmware causes fuzzer unable to cover part of the firmware. Fourth, we identified that not only the input values, but also the input duration (i.e., how long an input value/signal maintains), affect firmware execution on embedded devices. However, neither P^{2}IM nor any existing fuzzer considers input duration, which poses an open challenge for future research. For example, Soldering Iron and Reflow Oven firmware constantly read from GPIO while performing different operations determined by the duration of the same GPIO value/signal. Existing fuzzers generate inputs without considering input durations. Despite the existence of a functioning timer in P^{2}IM (by which the firmware can measure the duration of a GPIO value), firmware operations triggered by long durations of GPIO values can not be executed. The lack of support for varying input duration, although can be partly mitigated by P^{2}IM, needs to be better addressed by fuzzers, which are the dedicated component for input generation. We believe that input duration is a unique challenge for MCU firmware fuzzing, which is out of the scope of this paper and is an interesting topic to study for future works.

We also identified two reasons as to why we could not detect even more bugs than the 7 reported in Table 5. First, some firmware designs are not susceptible to memory corruption errors. For example, the Reflow Oven and Steering Control firmware rarely use buffers or dynamically allocated memory for performance reasons. Therefore, such firmware is unlikely to have memory corruption bugs. Second, the bare minimum memory error detector (or sanitizer) that we implemented may fail to catch the bugs triggered by the fuzzer. For example, we manually identified 1 extra bug on the PLC firmware. The bug, although similar to those found during fuzzing (Table 5), was triggered by P^{2}IM during testing but went undetected by the simple memory error detector.

6 Discussion

Direct Memory Access (DMA): P^{2}IM models the processor-peripheral interfaces, including registers and interrupts. It does not model Direct Memory Access (DMA), which allows peripherals to directly access RAM and in turn provide input to firmware. The lack of DMA support is a limitation of our work. Due to DMA's complex and peripheral-specific nature, modeling DMA is arguably impossible without considering internal peripheral designs, which goes against P^{2}IM's design principle—being generally applicable to a wide range of peripherals and MCU devices. Nonetheless, the usage of DMA depends on the design and architecture of individual firmware. We observed that most of the MCUs studied and tested support DMA. However, only 1 out of the 10 real firmware tested in §5.2 actually uses DMA.

Architectures beyond ARM: We analyzed 3 MCUs that use non-ARM architectures for IoT devices: ATmega328P (AVR), PIC32MX440F256H (MIPS) and FE310-G000 (RISC-V). Our analysis shows that our design of P^2IM and the abstract model that we defined are not specific to the ARM architecture. They can be extended to support the other architectures such as AVR, MIPS and RISC-V. All these non-ARM architectures define specific memory-mapped areas for peripherals similar to ARM. They also follow similar register categories (`CR`, `SR`, `DR` and `C&SR`) that P^2IM identifies. Furthermore, the procedures to configure and operate peripherals on these non-ARM architectures follow the same conventions and patterns that P^2IM uses for recognizing and handling peripheral I/O.

We observed a slight difference in accessing memory-mapped I/O by AVR. AVR can use either specific opcodes (IN/OUT) or ST/LD instructions to access mapped peripheral registers. ST and LD instructions require a constant offset to access the same addresses accessed by IN/OUT opcodes. We also observed that RISC-V implements a unique interrupt handling mechanism that uses Hardware Threads (HART) and a Platform-Level Interrupt Controller (PLIC). RISC-V also uses a new type of hybrid register (`S&DR`) that is not seen on the other architectures. P^2IM and the current abstract model can be extended to handle these architectural differences and in turn support these non-ARM MCU architectures.

Firmware Analysis beyond Fuzzing: Although our work was initially inspired by the open challenges facing firmware fuzzing, it is not designed to support fuzzing exclusively. Other types of dynamic firmware analysis that do not require fully accurate output from firmware can use our framework to achieve hardware-independence and scalability. For instance, data or code reachability analysis, such as taint analysis and certain debugging tasks, can benefit from our framework. In particular, concolic firmware execution can use our framework to generate more realistic concrete inputs (i.e., non-crashing/stalling), reduce the number of symbolic values, and avoid some infeasible code paths.

7 Related Work

Dynamic Firmware Analysis: Several recent works addressed the high barrier of dynamically analyzing MCU firmware. They follow the hybrid emulation approach, which forwards peripheral operations to real devices while running firmware on a customized emulator. Avatar [51] proposed a novel framework for hybrid emulation and used it for conducting concolic execution [30]. Surrogates [40] significantly improves the forwarding performance of Avatar via customized hardware. A follow-up work [43] fuzz-tested simple programs with manually injected vulnerabilities using Avatar and revealed that, without an effective sanitizer for MCU, fuzzers by themselves cannot observe many bugs even if they are triggered. Avatar2 [42] extends Avatar to allow replay of forwarded peripheral I/O without using real devices. Charm [48] targets smartphone drivers, rather than MCU firmware. It adopts the forwarding approach similar to Avatar. Prospect [39] forwards peripheral accesses at the syscall level, which however does not exist on bare-metal MCU devices. [38] uses cached peripheral accesses to approximate firmware states for analysis.

These works collectively improved the state of the art of dynamic firmware analysis. However, they have heavy hardware dependence, which is at odds with speedy and scalable fuzzing. Plus, they require significant expert-knowledge and human efforts to set up and run. In contrast, our framework is largely automated and completely removes hardware dependency from dynamic firmware analysis, without using peripheral I/O forwarding or replaying. Moreover, we found bugs from real MCU firmware whereas none of the previous works did, echoing the value of the scalable fuzzing enabled by our framework.

Another line of works analyzed firmware running in fully emulated environments [29, 33, 45] or directly on the target hardware [47]. Instead of MCU devices, these works target Linux-based devices, which are closer to general-purpose computers than truly embedded devices. Linux-based devices have much better emulator support due to the much less diverse peripherals than MCU devices. Analyzing firmware of these devices does not face the MCU-specific challenges that our work overcame.

Static Firmware Analysis: FIE [34] applied symbolic execution to TI MSP430 firmware by extending KLEE [28] with a peripheral model. It returns an unconstrained symbolic value for each peripheral register read. It assumes any enabled interrupts can happen after every instruction, which caused the state explosion problem. Inception [31] address this problem by optionally forwarding peripheral access to hardware using Avatar [51]. FirmUSB [36] proposed a symbolic execution mechanism tailored for USB controller firmware on 8051/52 architectures using domain specific knowledge. Firmalice [46] aims to find authentication bypass vulnerabilities in firmware via concolic execution. A large-scale study on Linux-based firmware [32] reported presence of weak passwords and known-vulnerable code. Although addressing the same high-level problem of firmware security, these works follow an orthogonal approach from ours (static vs. dynamic analysis).

8 Conclusion

We presented P^2IM, a novel technique for modeling the I/O behaviors of the processor-peripheral interfaces. It is the first to enable peripheral-oblivious emulation of MCU devices, and in turn, allow MCU firmware to be dynamically tested with high code coverage, at scale, and without hardware dependence. We built P^2IM into a framework that executes a given firmware binary and hosts a drop-in fuzzer (AFL) as the input source. We evaluated the framework using

70 sample firmware and 10 real device firmware. It fully booted and tested 79% of the firmware without any human intervention. When paired with a limited memory error detector, it found 7 new bugs from the real device firmware. The results suggest that our framework is of great value and potential for practical use.

References

[1] Building the krmx01 cnc. http://www.kronosrobotics.com/krmx01/.

[2] Can bus relay module. https://www.blinkmarine.com/can-bus-relay/.

[3] Cnc grbl stm32f4 source code. https://github.com/deadsy/grbl_stm32f4.

[4] Controllino maxi modbus library. https://github.com/CONTROLLINO-PLC/CONTROLLINO_Library/tree/master/MAXI.

[5] Controllino maxi plc. https://controllino.biz/controllino/maxi.

[6] Firmata library. https://github.com/firmata/arduino.

[7] Grbl. https://github.com/gnea/grbl/wiki.

[8] Heat press. https://en.wikipedia.org/wiki/Heat_press.

[9] Leading mcu suppliers. https://epsnews.com/2017/05/01/nxp-tops-microcontroller-supplier-ranking/.

[10] Modbus specification and implementation guides. http://www.modbus.org/specs.php.

[11] Patient monitor. http://www.utasco.com/en/monitoring-patsienta-2/patient-monitor.

[12] Qemu operating modes. https://qemu.weilnetz.de/doc/qemu-doc.html#Introduction.

[13] Quad-copter drone source code. https://github.com/heethesh/eYSIP-2017_Control_and_Algorithms_development_for_Quadcopter.

[14] Reflow oven. https://en.wikipedia.org/wiki/Reflow_oven.

[15] Reflow oven source code. https://github.com/rocketscream/Reflow-Oven-Controller.

[16] Riot os console. https://github.com/RIOT-OS/RIOT/tree/master/examples/default.

[17] Sainsmart toolpac pro32. https://www.amazon.com/SainSmart-ToolPAC-Soldering-Heating-Intelligent/dp/B01FFPE0EG.

[18] Self-balancing robot source code. https://github.com/mbocaneg/Inverted-Pendulum-Robot.

[19] Soldering iron source code. https://github.com/Ralim/ts100.

[20] Steer by wire. https://en.wikipedia.org/wiki/Drive_by_wire.

[21] Steering control source code. https://github.com/jabelone/car-controller.

[22] Stm32l010f4 microcontroller. https://www.st.com/resource/en/datasheet/stm32l010f4.pdf.

[23] Tiny reflow controller v2. https://www.rocketscream.com/blog/product/tiny-reflow-controller-v2.

[24] Understanding setpoint ramping and ramp/soak temperature control. https://www.west-cs.com/news/understanding-setpoint-ramping-and-rampsoak-temperature-control/.

[25] Unsecured ip camera list. https://reolink.com/unsecured-ip-camera-list.

[26] Why is traditional it security failing to protect the iot? https://www.timesys.com/security/traditional-it-security-failing-to-protect-iot.

[27] Michael Barr. Bookout v. toyota, 2005 camry l4 software analysis. http://www.safetyresearch.net/Library/BarrSlides_FINAL_SCRUBBED.pdf.

[28] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

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

[30] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[31] 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)*, 2018.

[32] Andrei Costin, Jonas Zaddach, Aurélien Francillon, Davide Balzarotti, and Sophia Antipolis. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, 2014.

[33] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[34] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, 2013.

[35] Drona Aviation. Pluto drone. https://www.dronaaviation.com/, 2017.

[36] Grant Hernandez, Farhaan Fowze, Dave Jing Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

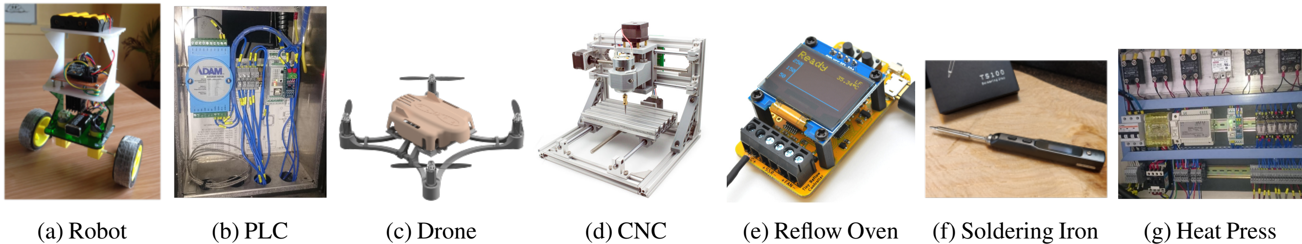
[37] Jesse Hertz and Tim Newsham. Triforceafl. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/.

[38] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. Embedded security testing with peripheral device caching and

Table 7: Real firmware tested in §5.2 and §5.4

Firmware	MCU	OS/Sys lib.	LoC/ELF file size	Source/Product	Product Image
Robot	STM32F103RB	Bare metal	32,999/960KB	[18] / DIY	Fig. 8a
PLC	STM32F429ZI	Arduino	10,578/774KB	[4]* / [5]	Fig. 8b
Gateway	STM32F103RB	Arduino	12,655/917KB	[6] / DIY	DIY
Drone	STM32F103RB	Bare metal	11,163/425KB	[13] / [35]	Fig. 8c
CNC	STM32F429ZI	Bare metal	7,561/287KB	[3] / [1]	Fig. 8d
Reflow Oven	STM32F103RB	Arduino	12,272/820KB	[15] / [23]	Fig. 8e
Console	MK64FN1M0VLL12	RIOT	6,984/1,132KB	[16] / DIY	DIY
Steering Control	SAM3X8E	Arduino	4,749/276KB	[21] / DIY	DIY
Soldering Iron	STM32F103RB	FreeRTOS	43,928/491KB	[19] / [17]	Fig. 8f
Heat Press	SAM3X8E	Arduino	4,150/248KB	[4]*/ Proprietary	Fig. 8g

*Only open-source libraries are disclosed, PLC/Machine control routines are property of their respective owners.



(a) Robot (b) PLC (c) Drone (d) CNC (e) Reflow Oven (f) Soldering Iron (g) Heat Press

Figure 8: Product images of the 10 real firmware

runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[39] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[40] Karl Koscher, Tadayoshi Kohno, and David Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT*, 2015.

[41] ARM Limited. *ARM®v7-M Architecture Reference Manual*, chapter B3.1.

[42] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *BAR*, 2018.

[43] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[44] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *International Conference on Privacy, Security and Trust (PST)*, 2015.

[45] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *OSDI*, 2012.

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

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[48] 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 *USENIX Security Symposium*, 2018.

[49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[50] Wikipedia. Stuxnet. https://en.wikipedia.org/wiki/Stuxnet, 2010.

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

[52] Michal Zalewski. Afl. http://lcamtuf.coredump.cx/afl/.

A Firmware Information

We present in Table 7 the detailed information about the real firmware used in the end-to-end test (§5.2) and fuzzing (§5.4).