A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers

Chongqing Lei[†], Zhen Ling[†*], Yue Zhang[‡], Yan Yang[†], Junzhou Luo[†], Xinwen Fu[§]
[†] *Southeast University, Email: {leicq, zhenling, yanyang, jluo}@seu.edu.cn*
[‡] *Drexel University, Email: zyueinfosec@gmail.com*
[§] *University of Massachusetts Lowell, Email: xinwen_fu@uml.edu*

Abstract

The extensive integration of embedded devices within the Internet of Things (IoT) has given rise to significant security concerns. Various initiatives have been undertaken to bolster the security of these devices at the software level, involving the analysis of MCU firmware and the implementation of automatic MCU rehosting methods. However, existing hardware-oriented rehosting techniques often face scalability challenges, while firmware-oriented approaches may have limited universality and fidelity. To address these limitations, we propose PERRY, a system that synthesizes faithful and extendable peripheral models for MCUs. By extracting peripheral models from hardware drivers, PERRY ensures compatibility and accurate emulation of targeted MCUs. The process involves gathering hardware metadata, analyzing driver code, capturing traces of peripheral accesses, and converting software beliefs into hardware behaviors. PERRY is implemented with approximately 19,000 lines of code. A comprehensive evaluation of 75 firmware samples has showcased its effectiveness, consistency, universality, and scalability in generating hardware models for MCUs. PERRY can efficiently synthesize hardware models consistent with the actual hardware and achieve a 74.24% unit test passing rate, outperforming the state-of-the-art techniques. The hardware models produced by PERRY can faithfully emulate diverse firmware and can be readily expanded with minimal manual intervention. Through case studies, we show that PERRY can help reproduce firmware vulnerabilities, discover specification-violation bugs in drivers, and fuzz RTOS for vulnerabilities. These case studies have led to the identification of two specification-violating bugs and the discovery of seven new vulnerabilities, underscoring PERRY's potential to enhance various security-focused tasks.

1 Introduction

The advent of the Internet of Things (IoT) has led to the extensive integration of embedded devices in various domains such as smart homes and industrial IoT, enabling enhanced connectivity and functionality. These embedded devices predominantly utilize microcontroller units (MCUs) to strike a balance between power efficiency and computational capabilities. However, in the context of extensive deployment of MCU-based IoT devices, ensuring their security is of utmost importance. These interconnected devices are susceptible to numerous vulnerabilities and security threats (e.g., unauthorized access, data breaches, and malicious attacks), which jeopardize the confidentiality, integrity and availability of sensitive information, presenting substantial risks to individuals and organizations [23, 30, 36].

Efforts have been made to bolster the software-level security of embedded devices, primarily through MCU firmware analysis [5, 9, 15, 31, 40, 44, 56, 61]. Among various analysis techniques, automatic MCU rehosting has emerged as a popular method. It involves running firmware in a virtual environment separate from the original hardware platform, allowing analysts to employ advanced dynamic analysis techniques and attain MCU platform independence. However, hardware-oriented rehosting techniques [9, 31, 56] rely on the actual hardware platform, employing hardware-in-the-loop emulation or record-and-replay mechanisms, which pose scalability challenges. Conversely, firmware-oriented rehosting techniques [5, 15, 44, 61], such as register-level approaches, infer peripheral responses based on register accesses but have limited universality and fidelity. Given these inherent limitations in the current state of research, there is an urgent demand for further advancements in this domain.

However, the task of creating accurate and adaptable peripheral models for MCUs is an exceedingly complex undertaking. It necessitates seamless integration, compatibility, and precise emulation of specific MCUs, all while grappling with the intricacies of hardware intricacies. Fortunately, we have observed that hardware is consistently accompanied by corresponding hardware drivers, which enable firmware to effectively access and utilize the hardware. These hardware drivers explicitly express expectations on corresponding hardware behaviors (i.e., software beliefs), which encompass crucial information about the behavior of the underlying hardware. This approach brings forth significant advantages since the drivers inherently conform to the underlying hardware for normal operation, ensuring that the inferred hardware models are universally applicable and accurate.

Building upon this observation, we introduce PERRY, a comprehensive methodology comprised of four essential

[*]Corresponding author: Prof. Zhen Ling of Southeast University, China.

steps. First, we collect hardware metadata, acquire the driver source code, compile it into LLVM bitcode, and extract supplementary information using Clang/LLVM. Second, we analyze the driver bitcode, establish a symbolic execution environment, and capture traces of peripheral accesses through symbolic execution. Third, from these traces, we extract software beliefs about the underlying hardware and convert them into hardware behaviors. Finally, by using the inferred hardware behaviors and user inputs, we fill in the gaps within the hardware model template, resulting in a complete and valid hardware model. This synthesized hardware model can be seamlessly integrated into an emulator such as QEMU, enabling the emulation of diverse firmware on the target MCU.

During the design of PERRY, we face several challenges, all of which are successfully addressed. For instance, we introduce a set of rules to identify software beliefs within drivers and convert these beliefs into hardware behaviors. Additionally, we tackle the challenge of efficiently analyzing driver libraries through symbolic execution by breaking loops and eliminating scoped constraints.

We implement PERRY by leveraging Clang/LLVM [33], KLEE [4], and Z3 [11], incorporating approximately 19,000 lines of our own code. We conduct a comprehensive evaluation of PERRY using 75 firmware samples, assessing its efficiency, consistency, universality, and scalability. Evaluation results on 10 driver libraries show that PERRY can infer hardware behaviors efficiently for over 30 MCUs. These behaviors are consistent with actual hardware, yielding a 74.24% unit test passing rate and surpassing the state-of-the-art. Hardware models generated by PERRY can be directly used to emulate various firmware with high fidelity, and can be easily extended to support more non-trivial hardware functionalities with minimum manual efforts.

We conduct three case studies to illustrate the potential security applications of PERRY. In the first study, we demonstrate that hardware models generated by PERRY could rehost BLE host firmware, facilitating the replication of BLE protocol stack vulnerabilities. In the second case study, we demonstrate how hardware models inferred from drivers could be cross-referenced with corresponding hardware specifications, allowing us to identify deviations or violations within the drivers. Finally, in the third case study, we fuzz real-time operating systems (RTOS) and uncover vulnerabilities by emulating RTOS with PERRY-generated hardware models. Consequently, we discover two specification-violation bugs in drivers and seven new vulnerabilities in RTOS, underscoring the versatility and practicality of PERRY in the realm of security.

Our major contributions are summarized as follows.

- **Novel Insights.** We demonstrate that drivers expose information about the behaviors and characteristics of the underlying hardware, which can be effectively extracted and processed to infer actual hardware behaviors.
- **Practical Tool.** We introduce PERRY, which collects hardware metadata and driver source code, analyzes the driver's LLVM bitcode to capture traces of peripheral access, extracts hardware behaviors from these traces for symbolic execution, and subsequently completes the hardware model template with the inferred behaviors.
- **Extensive Evaluations.** We perform a thorough evaluation of PERRY with 75 firmware samples to demonstrate its efficiency, consistency, universality, and scalability. PERRY can efficiently synthesize faithful hardware models and achieve a 74.24% unit tests passing rate, outperforming the state-of-the-art. PERRY-generated hardware models can be used to emulate diverse firmware with minimum manual efforts.
- **Security Implications.** Our security case studies demonstrate the capability of PERRY in replicating BLE protocol stack vulnerabilities, detecting specification-violation bugs, and fuzzing RTOS for new vulnerabilities.

Availability. PERRY is open source at https://github.com/VoodooChild99/perry.

2 Background

2.1 MCU-based Embedded Systems

Architecture of MCU-based Embedded Systems. MCU-based embedded systems refer to resource-constrained devices built for specific purposes, utilizing MCUs as the core processing units. As shown in Figure 1, these systems run firmware, which contains the control logic and interacts with various peripherals through peripheral drivers. The firmware can be executed on top of an MCU with a lightweight RTOS or directly on the hardware (bare-metal). In these systems, the peripherals expose interfaces through registers that can be accessed by the CPUs using memory-mapped I/O (MMIO). By accessing these MMIO registers, the firmware configures working modes, acquires the working status of peripherals, and performs data exchanges with external environment, enabling a wide range of hardware-related functionalities.

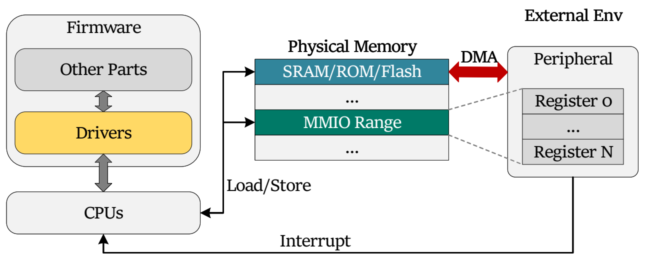

Figure 1: MCU-based embedded systems architecture.

At a high level, MCU drivers are the proxy between the software world and the hardware world. As such, they must translate software requests into hardware interactions, read hardware responses, and provide them back to the software. Software will then perform various tasks (e.g., processing protocol packets) utilizing these results.

MCU Peripherals. MCU peripherals, in addition to passive CPU access, utilize interrupts to notify the CPU of specific hardware events. For example, a UART triggers an interrupt to alert the firmware to read incoming data. Peripherals that support interrupts have distinct interrupt sources connected to the interrupt controller. When certain conditions in the registers are met, the corresponding interrupt is activated, and the signal is sent to the CPU if enabled. The CPU then pauses the ongoing code execution and jumps to the appropriate interrupt service routines (ISRs) based on the interrupt number. Since there is a finite number of interrupt numbers, multiple hardware events may share the same number, resulting in the same interrupt for different events. To specify the actual event, peripherals configure diverse register values to assist ISRs in demultiplexing interrupts. It's worth noting that peripherals can also interact directly with memory through direct memory access (DMA), thereby enhancing firmware throughput without requiring CPU involvement.

2.2 Automatic MCU Rehosting

Rehosting techniques are designed to execute firmware in a virtual execution environment, independent of the original hardware platform. This enables the examination of MCU firmware through sophisticated dynamic analysis techniques. Rehosting necessitates emulating CPUs and peripherals [14]. Emulating CPUs is labor-intensive but a one-time effort (e.g., QEMU [2]), given the relatively small amount of CPU models, this is acceptable in general. However, the number of peripherals is much larger than that of CPU models (e.g., over 30 peripherals per SoC) [14]. Although it takes relatively less manual efforts to model a peripheral, the huge amount of peripherals makes it impractical to emulate all of them. As a result, emulators often have few hardware models [14], and analysts have to implement missing models on their own. This makes automatic MCU peripheral emulation the key challenge in rehosting. In this section, we review the literature on rehosting techniques, which can be grouped into two categories based on their implementation.

- **Hardware-Oriented Rehosting.** Some techniques leverage the inherent capabilities of the hardware platform to accomplish MCU rehosting, and these approaches can be categorized into two groups. The first kind of rehosting technique is called hardware-in-the-loop [9, 31, 56], where only CPUs are emulated, and peripheral accesses are forwarded to real devices. In the second category of techniques [20, 49], peripheral interactions during firmware executions are recorded, peripheral outputs are analyzed and then replayed during firmware emulation. However, these techniques face scalability challenges due to their reliance on physical hardware.
- **Frimware-Oriented Rehosting.** Other techniques aim to provide different virtual execution environments for different firmware and can be divided into register-level approaches and function-level approaches. Function-level techniques [8, 34, 39] involve summarizing peripheral driver functions to remove hardware interactions and only retain semantics, but they are limited by the manual effort required and the lack of practicality due to the absence of debugging information in MCU firmware binaries. Register-level techniques [5, 15, 44, 61] infer how peripherals respond to register accesses to explore more firmware paths instead of faithfully emulating hardware, but they suffer from low universality (e.g., these models are firmware-specific and limited in emulating complex hardware functionalities), making them specific to analyzed firmware, and low fidelity, as they cannot accurately replicate actual hardware behaviors.

An concurrent effort [62] seeks to address the aforementioned limitations by creating highly accurate hardware models based on hardware specifications. However, this approach entails a significant amount of manual work, such as manually identifying expected hardware descriptions in manuals, converting these descriptions from PDF format to plain text, and diagnosing the generated rules. These manual tasks restrict its practicality.

3 Problem and Motivating Example

3.1 Threat Model and Problem Statement

Possible Threats. Malicious data can be injected into the firmware of a MCU through its peripherals. Vulnerabilities in the driver code of a peripheral may then be exploited [35]. The malicious data can propagate into other parts of the firmware such as upper-layer applications, whose vulnerabilities may also be exploited [23, 43].

Conducting security analysis on MCU firmware presents challenges due to significant influences of the hardware on both control flow (e.g., interrupts) and data flow. Emulating hardware often requires huge manual efforts, and emulators typically have few hardware models (see §2.2). Therefore, automating hardware emulation is essential.

Problem Objectives. Our objective is to autonomously generate extensible and accurate peripheral models for a target MCU to facilitate security analysis. The generated peripheral model faithfully reflects actual hardware models and can be easily integrated into existing emulators. As a result, these models can be used to rehost various firmware, thus enabling security analysis of MCU firmware.

However, this can be challenging. The integration of these peripheral models into existing emulators is a complex task, as ensuring seamless integration and compatibility between the generated models and the emulator framework can be technically demanding. It requires careful consideration of the interface, timing, and synchronization aspects to ensure the accurate emulation of the targeted MCU. Additionally, firmware variations, different software versions, and compiler optimizations further complicate the task of developing firmware-independent rehosting techniques. Lastly, the lack of comprehensive documentation, proprietary components, and closed-source firmware can hinder the reverse-engineering and understanding of hardware behaviors.

Problem Assumptions. We have the following prerequisites and assumptions. First, we assume that we can access the source code for at least one implementation of the peripheral driver. We believe that requiring the source code for the driver is reasonable: (i) MCU vendors do make driver source code public for MCU peripherals. (ii) Any open-source driver even one by a third party for specific hardware is sufficient. We conduct a survey on 10 top MCU vendors as shown in Table 1 and find that all investigated MCU vendors provide public access to driver source code.

Table 1: The availability of MCU driver source code

Vendor	Region	Host*	Example	First Release (YYYY/MM)
STMicroelectronics	Switzerland	GH	[51]	2019/04
NXP Semiconductors	Netherland	GH	[47]	2021/01
Microchip Technology	USA	OW	[54]	2022/10
Texas Instruments	USA	GH	[26]	2022/12
Renesas Electronics	Japan	GH	[12]	2019/10
Infineon Technologies	Germany	GH	[53]	2019/08
Nuvoton Technology	Taiwan, China	GH	[55]	2021/12
Nordic Semiconductor	Norway	GH	[46]	2017/10
Espressif Systems	Mainland China	GH	[52]	2016/08
GigaDevice Semiconductor	Mainland China	OW	[45]	2023/08

* GH: GitHub, OW: Official Website

Second, we assume the MCU corresponding to the target firmware is known. Such information can be obtained from the product brochure or model number printed on the SoC. It is needed for acquiring corresponding drivers and building hardware models. For example, Giese [16] identified several MCU models combing both methods.

Third, like previous efforts [15, 44], we assume that the hardware metadata of the target MCU is available, including memory mapping information, CPU model and initial interrupt vector location. The information can be extracted from hardware manuals. Specifically, for ARM Cortex-M MCUs, such information can be obtained more easily by parsing the Common Micro-controller Software Interface Standard System View Description (CMSIS-SVD) file [1] of the target MCU.

3.2 Motivating Example

As discussed in §3.1, synthesizing faithful and extendable peripheral models for MCUs is a complex task. However, we have made an observation that allows us to achieve our goal without the need for additional documentation processing or component analysis.

Key Insight. We observe that hardware is always accompanied by corresponding hardware drivers so that the hardware can be accessed and used by firmware, and these hardware drivers contain information about the behaviors of the underlying hardware. Based on this observation, to achieve our goal, we propose to extract peripheral models from hardware drivers. As a benefit, because drivers must respect the underlying hardware to function normally, hardware models inferred from drivers are universal and faithful.

We would like to illustrate this observation with a motivating example. Specifically, peripheral drivers must effectively manage hardware states to ensure successful interaction with the hardware and the realization of expected functionalities. Typically, this is accomplished through the use of various conditional statements that check the values of specific hardware registers. Listing 1 shows the code snippet taken from the universal asynchronous receiver/transmitter (UART) driver for STM32F7 series MCUs. The code has been simplified for the ease of reader comprehension. The API `HAL_UART_Receive()` is exposed to upper-layer applications for receiving incoming data through UART. According to the specification of STM32F7 UART peripheral [50], the `RXNE` bit of the `ISR` register is set by hardware when incoming data is ready to be read from the `RDR` register. Therefore, in correspondence with the specification, the driver: 1) waits until the `RXNE` bit is set in the `ISR` register (line 5), and then 2) reads the data register `RDR` for the incoming data and stores it into a buffer (line 9).

```
1  // uart.c
2  HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart,
   ↪  uint8_t *pData, uint16_t Size, uint32_t Timeout) {
3    ...
4    while (Size > 0) {
5      /* wait until UART_FLAG_RXNE flag in the ISR register is
       ↪  set by the hardware */
6      if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_RXNE,
       ↪  RESET, tickstart, Timeout) != HAL_OK) {
7        return HAL_TIMEOUT;
8      }
9      /* read incoming data from the TDR register */
10     *pData = (uint8_t)(huart->Instance->RDR & (uint8_t)uhMask);
11     ++pData;
12     --Size;
13   }
14   ...
15   return HAL_OK;
16 }
```

Listing 1: STM32F7xx UART driver code snippet.

Assuming we have no prior knowledge about UART behaviors when receiving data, we can still deduce this information from the driver code. First, by analyzing the code, we can know from the type (details will be elaborated upon in §4.1) that `huart->Instance` actually represents a peripheral, which means accessing its members will access MMIO registers. Second, by examining all the code paths, we observe that the data within the `RDR` register is written into a data buffer, implying that `RDR` serves as a data register. Third, after establishing this understanding, we can analyze the path constraints and observe that the `RDR` register is accessed only when the `UART_FLAG_RXNE` flag in the `ISR` register is set. This indicates that the flag should be set by the UART hardware when the incoming data in `RDR` is ready to be read – otherwise, the driver will never read the data. Finally, with the inferred UART behavior, we insert the following logic into our hardware model to mimic such hardware behavior: when the UART receives external inputs, we first move the data into the `RDR` register and then set the `UART_FLAG_RXNE` flag in the `ISR` register.

4 PERRY Design

Based on the example in §3.2, we can conclude that to synthesize peripheral models from drivers, we must process the driver source code to extract useful information (e.g., register accesses), observe how registers are accessed in drivers, infer hardware behaviors from the observed register accesses, and finally synthesize a hardware model using these inferred behaviors. As shown in Figure 2, we propose a system called PERRY[1] to synthesize MCU peripheral models from peripheral drivers, which can be divided into four phases:

① **Pre-Processing (§4.1).** We perform several steps to prepare the input required by the system. This includes gathering the hardware metadata and acquiring the driver source code (see §3.1). To facilitate further analysis, we compile the driver source code into LLVM bitcode and extract relevant auxiliary information using Clang/LLVM.

② **Trace Collection (§4.2).** With these inputs in hand, we proceed to analyze and instrument the driver bitcode to establish the symbolic execution environment correctly. Additionally, we employ symbolic execution to analyze the driver APIs and capture traces of peripheral accesses.

③ **Model Inference (§4.3)** Next, we extract software beliefs regarding the underlying hardware from the collected traces and convert these beliefs into hardware behaviors that support and justify them.

④ **Model Synthesis (§4.4).** Finally, we utilize the inferred hardware behaviors along with user inputs to complete the missing sections in the hardware model template, thereby synthesizing a valid hardware model. This generated hardware model can then be seamlessly integrated into the emulator, such as QEMU [2], enabling the emulation of various firmware running on the target MCU.

[1]The system is named PERRY because its pronunciation is reminiscent of the word "peri", which stands for "peripheral".

4.1 Pre-Processing

PERRY takes hardware metadata and the driver source code as inputs. It parses the names and memory ranges of peripherals to be analyzed and obtains the target hardware information from hardware metadata. When compiling the driver source code to LLVM bitcode, we collect three types of source-level information (**SI**): loop header location, success return values, and peripheral structure names.

(SI-I) Loop Header Location. Loop headers (e.g., line 3 in Listing 1) define loop iterations. Drivers typically employ a waiting strategy, repeatedly reading specific registers in loop headers, and anticipating hardware updates. These features greatly aid in inferring hardware models (See §4.3). "To facilitate model inference, we gather loop header locations from the driver source code during preprocessing in the format of "`(filename, beginLoc, endLoc)`". For example, in Listing 1, the loop header `(uart.c, {3, 3}, {3, 18})` at line 3 has `beginLoc` referring to "w" and `endLoc` referring to ")".

(SI-II) Success Return Values. Drivers often perform a preliminary check to ensure proper functionality of a peripheral before certain operations (e.g., reading the data register). If the check succeeds, normal operations are carried out (valid paths). If the check fails, the driver may immediately return an error status code and skip all subsequent operations (invalid paths). To accurately infer expected hardware behaviors and synthesize hardware models, focusing solely on valid paths and disregarding invalid paths is essential. To achieve this, we collect return values that indicate success, aiding PERRY in identifying valid paths. Drivers commonly utilize enums to denote execution status, which possess semantic features. For instance, the enum value `HAL_OK` (line 14 in Listing 1) signifies successful execution, while `HAL_TIMEOUT` (line 6 in Listing 1) indicates a timeout error. Based on this observation, we gather enum values returned by driver functions and identify success return values by their names. If a name contains typical words denoting success (e.g., `"ok"` or `"success"`), we consider the corresponding value as a success return value. We contend that this pattern sufficiently identifies success return values, as all drivers listed in §5 adhere to this convention.

(SI-III) Peripheral Structure Names. Peripheral registers are arranged in a compact and contiguous manner within the peripheral's address space. To abstract the layout of a peripheral, drivers commonly employ structures known as peripheral structures, where each structure member represents a register within the peripheral. Peripheral structure instances

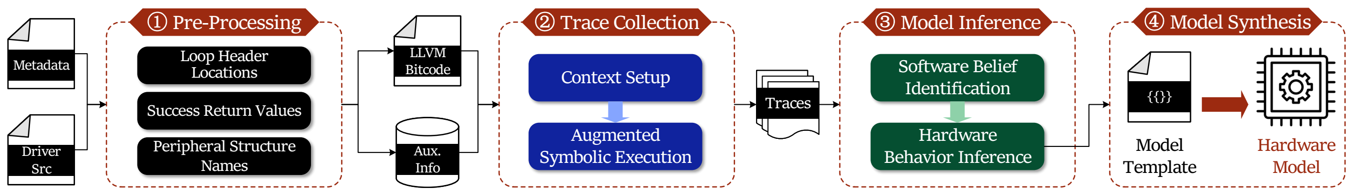

Figure 2: PERRY overview.

are typically hard coded and mapped to the actual peripheral address, and drivers directly use these instances to access peripherals. However, some peripherals, such as UART, may share the same peripheral structure. To ensure generality, drivers for such peripherals often avoid using hard-coded peripheral structure instances. Instead, they allow users to specify the instance through a parameter. In such cases, we need to identify these parameters and assign them to the corresponding peripheral structure instances. We have observed that structure instances are typically defined using macros that follow a common pattern, as shown below:

```
#define [INSTANCE] (([STRUCT]*) [ADDRESS])
```

where `[INSTANCE]` represents the instance name, `[STRUCT]` represents the structure name, and `[ADDRESS]` represents the peripheral address. For example, the macro below defines a peripheral instance `USART1` at the address `USART1_BASE` using the structure `USART_TypeDef`.

```
#define USART1 ((USART_TypeDef *) USART1_BASE)
```

By parsing macros in drivers, we can extract peripheral structure names accordingly.

4.2 Trace Collection

PERRY collects peripheral access traces during the symbolic execution of driver APIs. To this end, PERRY first analyzes and instruments the driver bitcode to properly setup the context for symbolic execution. After this, PERRY records essential peripheral access information through augmented symbolic execution.

4.2.1 Context Setup

Five tasks (**T**) must be performed to properly setup the symbolic execution context for drivers. First, we prepare MMIO regions to replicate the actual MCU memory layout. Second, we identify potential entry points for symbolic execution, as drivers are commonly found in library form. These identified entry points are then organized to establish their corresponding calling contexts. In the fourth task, we identify and taint input data buffers to aid in the recognition of data registers. Lastly, we identify and hook callbacks to aid the inference of interrupt conditions.

(T1) Prepare MMIO Regions. Peripheral drivers access peripheral registers by directly accessing specific memory addresses (i.e. MMIO regions). To enable the symbolic executor to handle register accesses, we pre-map these MMIO regions. Additionally, we need to track the usage of values obtained from registers to infer hardware models. To achieve this, we symbolize all mapped MMIO regions and allocate unique taint labels to different registers as taint sources.

(T2) Identify Entry Points. Driver code, unlike code with a pre-defined entry point like `main()`, is typically presented as a library. This means that multiple potential entry points (i.e., APIs) are exposed without being used. To pinpoint potential entry points for symbolic execution within drivers, we construct a call graph and identify top-level functions that are not invoked by other functions. We identify top-level functions as entry points for two reasons: firstly, top-level functions typically encapsulate complex peripheral functionalities and engage in extensive hardware interactions; secondly, interactions between different top-level functions with hardware are often non-interfering, ensuring comprehensive observation of hardware behavior.

(T3) Assemble Entry Points. We set up the program environment (i.e., parameters and global variables) for each API to enable symbolic execution as they are not directly used. The entry point assembly algorithm is shown in Algorithm 1. Parameters and global variables are analyzed and prepared following the principle of avoiding symbolic pointers. For pointer variables, we recursively allocate an array of objects of the underlying type and symbolize their content (line 21-26). The array length is typically set to 1, unless it is a data buffer (see **T4**) where a larger constant value is used (e.g., 8). Non-pointer variables are allocated as objects with their content symbolized (line 29-30). If the object is a structure, we repeat the above procedure for each field (line 31-35). Two exceptions are pointers to peripheral structures and constant global variables. Pointers to peripheral structures are assigned the corresponding concrete peripheral physical addresses (line 12). Constant global variables are not made symbolic as their values remain unchanged. The above method fails to handle function pointers. Inspired by MLTA [38], we adopt a light-weight yet effective type analysis to determine indirect call targets (line 14).

(T4) Taint Input Data Buffers. Driver APIs that send data to peripheral hardware typically use data buffers to store user

Algorithm 1: Entry Point Assembly Algorithm

```
Input  : Set of identified entry points $S_{API}$
Output: Prepared program environment for symbolic
        execution
1  foreach $API \in S_{API}$ do
2      SetupProgramEnvironment($API$);
3  end
4  Function $SetupProgramEnvironment(API)$:
5      $S_{para}, S_{glob} \leftarrow$AnalyzeParametersAndGlobals($API$);
6      foreach $var \in S_{para} \cup S_{glob}$ do
7          SetupVariable($var$);
8      end
9  Function $SetupVariable(var)$:
10     if $isPointer(var)$ then
11         if $isPeripheralStructPointer(var)$ then
12             AssignConcreteAddress($var$);
13         else if $isFuncPtr(var)$ then
14             FillinFuncPtrByType($var$);
15         else
16             AllocateAndSymbolizeArray($var$);
17         end
18     else
19         AllocateAndSymbolizeObject($var$);
20     end
21 Function $AllocateAndSymbolizeArray(ptrVar)$:
22     $pointee \leftarrow$ getPointee($ptrVar$);
23     $len \leftarrow$ isDataBuffer($ptrVar$) ? $CONSTANT$ : 1;
24     for $i = 1$ to $len$ do
25         SetupVariable($pointee$);
26     end
27 Function $AllocateAndSymbolizeObject(var)$:
28     if $\neg isConstant(var)$ then
29         AllocateObject($var$);
30         SymbolizeContent($var$);
31         if $isStruct(var)$ then
32             for $field \in getFields(var)$ do
33                 SetupVariable($field$);
34             end
35         end
36     end
```

inputs, which are directly written into data registers. Based on this observation, we adopt a taint-based mechanism to recognize data registers. We first identify data buffers based on API parameter types and names. If a parameter has a buffer type (e.g., `uint8_t *`) and its name matches common patterns (e.g., `"data"`), it is considered a data buffer. We then taint the content of each identified data buffer to track data flows from user data to peripheral registers. By analyzing the taints within registers, we can identify data registers where the taints come from data buffers.

(T5) Hook Callbacks. Interrupt handlers frequently trigger callbacks upon successfully managing interrupts. By intercepting the callbacks and collecting the associated path constraints (see §4.3.1), PERRY can infer interrupt conditions. Callbacks are recognized as unresolved function pointers or empty functions with weak linkage. For each identified callback, we reroute it to our built-in hook functions.

4.2.2 Symbolic Execution for Trace Collection

PERRY combines symbolic execution and taint analysis for hardware model inference. Similar to previous efforts [5, 44, 61], peripheral registers are set as non-volatile symbol/taint sources. However, two challenges arise due to the limitations of symbolic execution and unique characteristics of drivers. Firstly, symbolic execution can lead to state explosion due to extensive loop usage in drivers. Secondly, symbolic execution can be blocked due to conflicting constraints caused by hardware. Drivers initially assume a specific register value, which is then updated by hardware. Afterwards, drivers might assume a different value for the same register. However, the two constraints conflict with each other and symbolic execution is blocked. To address these challenges, we use two methods (**M**): actively breaking loops during symbolic execution to reduce complexity, and utilizing checkpoints to remove hardware-related constraints. These mechanisms efficiently achieve the goal (**G**) of collecting peripheral register access traces and execution results for further processing.

(M1) Break Loops for Path Pruning. To facilitate effective exploration of software-hardware interactions, we employ loop handling that imposes constraints on program states rather than terminating them. This approach enables continued program execution by detecting and exiting loops. When the symbolic executor encounters a conditional branch instruction, it checks for repeated execution of the branch target beyond a predefined threshold (e.g., twice). If exceeded, it negates the loop condition and jumps out of the loop, allowing uninterrupted execution.

```
1 while(__HAL_RCC_GET_FLAG(RCC_FLAG_PLLRDY) != RESET) { ... }
2 ...
3 while(__HAL_RCC_GET_FLAG(RCC_FLAG_PLLRDY) == RESET) { ... }
```

Listing 2: Conflicting constraints due to symbol reuse.

(M2) Remove Constraints using Check-Points. Listing 2 shows an example of conflicting hardware-related constraints caused by reusing the same symbol. If the constraint at line 1 evaluates to false, the constraint on line 3 will always evaluate to true, and the driver API will return an error because we cannot break such a loop – the negated loop condition will conflict with existing path constraints. While on a real device, both constraints can be satisfied since the accessed register is updated by the hardware before line 3. We observe that such constraints (e.g., line 1) usually appear as scoped loop conditions, i.e., their impacts on program execution is restricted to be within the loop body. Based on this observation, we propose a check-point-based mechanism to detect and remove these scoped hardware-related constraints.

A check point is defined as a (PC, CS) pair, where CS denotes a hardware-related constraint, and PC denotes the code location where CS is introduced. The check point mechanism maintains a set of check points within the current function. When a hardware-related constraint evaluates to true, we check if the corresponding $(PC\prime, CS\prime)$ pair exists in the set. If so, we identify $CS\prime$ as a scoped hardware-related constraint, remove it from the current program state, and force the program to take the negated branch.

(G) Trace Collection. For each explored program state, we collect the following information.

- **Exit Status (*ES*)**: Program states can terminate normally (i.e., reaching the end of an execution path) or abnormally (e.g., errors during symbolic execution). Traces from normally exited program states provide valuable information for hardware model inference, while traces from other program states are less useful.
- **API Return Value (*RV*)**: Program states with success return values provide valuable insights into the expected hardware behavior.
- **Path Constraints (*CS*)**: The path constraints until program state terminates. These constraints help PERRY infer hardware behaviors.
- **Register Accesses (*RA*)**: During symbolic execution, the recorded information for each identified peripheral access includes: (*i*) access action A (read or write); (*ii*) accessed offset O within the peripheral; (*iii*) access width W (e.g., 4 bytes); (*iv*) symbolic expression S returned for a read or to be written for a write; (*v*) taint T of the value to be written (only for writes); (*vi*) path constraints C until the access; and (*vii*) code location L of the access.
- **Output Data Buffer Taints (*OT*)**: Tainting input data buffers helps identify data registers (see §4.1). However, for peripherals with separate registers for receiving and transmitting data, the previous mechanism only identifies transmitting data registers. To address this, we additionally collect taints (OT) from data buffers after executing the API. If a taint from a register is found in OT, the register is recognized as a receiving data register.
- **Callback Hook Invocation Constraints (*CC*)**: The path constraints when the callback hook is invoked. These constraints help PERRY infer interrupt conditions.

4.3 Model Inference

With the collected driver traces, we perform model inference, which involves identifying basic software beliefs and convert them into hardware behaviors (§4.3.1), and identifying complex software beliefs and transforming them into hardware behaviors using on-demand analysis (§4.3.2).

4.3.1 Basic Software Beliefs Identification and Hardware Behaviors Transformation

Software beliefs represent software's expectations on hardware, i.e., hardware is believed to perform certain functionalities after instructions issued by software. The most basic beliefs reside in conditional statements checking whether hardware has reached expected states. For example, checking status registers before accessing peripheral functionalities (e.g., read incoming data) to guarantee the success of subsequent operations, or after configuring peripherals (e.g., enable a clock) to indicates the success of previous configurations. These checks represent the software's assumptions regarding expected hardware behaviors and can be employed to infer actual hardware behavior. We categorized four distinct basic types of software beliefs (**SB**) related to anticipated hardware behaviors:

- **Reading data registers (RDR)**: These beliefs are derived from constraints related to data register reads.
- **Writing data registers (WDR)**: These beliefs stem from constraints related to data register writes. We observe three sub-kinds of WDR beliefs: the first write (FW), between two writes (BW) and after all writes (AW).
- **Handling interrupts (HINT)**: These beliefs are drawn from constraints associated with interrupt handling.
- **Updating non-data registers (UNDR)**: These beliefs are inferred from correlations among non-data registers (e.g., updating one register leads to the software waiting for another to be updated).

Software may also express beliefs using other statements. For example, drivers may store the variable representing source address into the source address register of a DMA peripheral, under the belief that the register holds DMA transmission source address. Unlike the above SBs that can be located by conditional statements, such beliefs are harder to locate as they also involve semantic information of drivers. We therefore equip PERRY with the on-demand analysis capability to handle such complex SBs (see §4.3.2). We demonstrate how to transform basic SBs into hardware behaviors below.

RDR Transformation. RDR beliefs capture the constraints that software should follow to successfully read incoming data from data registers. Speculatively, when the hardware receives incoming data and updates the corresponding data registers, it should satisfy the constraints specified in RDR beliefs. This enables the software to detect the signal and read the incoming data. To analyze the hardware behavior, we group RDR beliefs based on the accessed data register (using offset O). Then, we derive the hardware behavior for receiving incoming data through the data register by combining all the constraints using disjunction. Formally, for a group of RDR beliefs $\{(C_i, Sym_O) \,|\, i = 1, ..., N\}$, we derive a hardware behavior:

$$B_{RDR} := \bigvee_{i=1}^{N} C_i \tag{1}$$

WDR Transformation. WDR beliefs capture the constraints that software should follow to successfully send data through data registers. While there are three sub-kinds of WDR beliefs, it can be assumed that the constraints in WDR beliefs must evaluate to true after user data is written into a data register. Otherwise, the data transmission process in the software would fail. To analyze the hardware behavior, we group WDR beliefs for each sub-category by the accessed data register. This results in a triple of WDR belief groups (G_{FW}, G_{BW}, G_{AW}) for each data register corresponding to three sub-kinds. For each triple, we derive the hardware behavior when transmitting data through the data register by disjuncting constraints within each triple element, then conjuncting the results. Formally, assume $G_{FW} = \{(C^i_{FW}, Sym_O) \,|\, i = 1, ..., N\}$, $G_{BW} = \{(C^i_{BW}, Sym_O) \,|\, i = 1, ..., M\}$, and $G_{AW} = \{(C^i_{AW}, Sym_O) \,|\, i = 1, ..., K\}$, we derive the following hardware behavior:

$$B_{WDR} := \left(\bigvee_{i=1}^{N} C^i_{FW}\right) \wedge \left(\bigvee_{i=1}^{M} C^i_{BW}\right) \wedge \left(\bigvee_{i=1}^{K} C^i_{AW}\right) \tag{2}$$

HINT Transformation. HINT beliefs consist of path constraints that exist before executing interrupt handling operations in ISRs. We regard data register reads/writes and callback hook invocations as interrupt handling operations. These constraints include triggering conditions for different hardware events. Since the interrupt condition is composed of these hardware event triggering conditions, we can infer the actual interrupt triggering condition by combining all the HINT belief constraints using disjunction. In other words, the interrupt will be triggered if at least one event is triggered. Formally, assuming $\{C_i | i = 1, ..., N\}$ represents all the HINT beliefs, we derive the following hardware behavior:

$$B_{HINT} := \bigvee_{i=1}^{N} C_i \tag{3}$$

UNDR Transformation. UNDR beliefs describe the relationship between two non-data registers. If a register Reg_A is written in a specific way, it is expected that the hardware will update another register Reg_B accordingly. To translate UNDR beliefs into hardware behaviors, we need to consider the write semantics (how Reg_A's value is changed) and the update semantics (how Reg_B should be updated). In most cases, the value written into Reg_A is bounded and the expected value for Reg_B can be easily derived from the path constraints. We propose a bit value variation-based approach to infer write semantics under such cases. However, sometimes the value written into Reg_A is unbounded, making the above method fail to infer the write semantic. To handle such cases, we additionally propose a write-update dependency resolution method to synthesize a linear formula between the written value and the expected update value. The technical details of the proposed two methods can be found in Appendix A. With the inferred write semantic WS and update semantic US, the hardware behavior can be derived as:

$$B_{UNDR} := (WS, US) \tag{4}$$

4.3.2 On-Demand Complex Software Beliefs Identification and Hardware Behaviors Transformation

PERRY supports on-demand analysis to locate complex SBs and transform them into hardware behaviors. Analysts can use commands to instruct PERRY to identify complex SBs. PERRY will then transform these beliefs into hardware behaviors accordingly. PERRY recognizes three types of commands from analysts – `PARAM`, `CALLBACK` and `FUNCTION`.

- **PARAM:** `PARAM` commands describe the software belief where specific data is supposed to be stored into dedicated registers. A `PARAM` command contains parameters of a function. PERRY handles such commands by locating all registers where the specified parameters are written into, and outputs these registers as hardware behaviors.
- **CALLBACK:** `CALLBACK` commands describe the software belief where certain hardware events are supposed to happen. A `CALLBACK` command contains the name of a callback. PERRY handles `CALLBACK` commands by gathering path constraints when the specified callback is invoked, and generates a hardware behavior expressed as the disjunction of these constraints.
- **FUNCTION:** `FUNCTION` commands describe the software belief where certain hardware functionalities should be triggered by invoking the function. A `FUNCTION` commands contains the name of a function. PERRY handles `FUNCTION` commands by locating all register updates within the specified functions, and generates hardware behaviors based on their write semantics.

4.4 Model Synthesis

The last step of PERRY is to synthesize a hardware model that can be directly integrated into emulators using the inferred hardware behaviors. To this end, we adopt a template-based synthesis approach. PERRY first takes user-provided hardware metadata file and generate a basic hardware model template file, where only basic hardware behaviors are included – register reads return the current register value, and register writes will update corresponding registers. Other hardware behaviors are left as holes, which are then synthesized and filled based on the inferred hardware behaviors. We demonstrate the synthesis procedure for behaviors inferred from basic SBs (basic behaviors) and complex SBs (complex behaviors) respectively.

Basic Behaviors. There are four kinds of holes corresponding to the four basic behaviors, we reuse the name of corresponding belief as introduced in §4.3.1 as the name of each kind of holes for simplicity. For RDR holes, we synthesize a function that takes inputs from the host, store inputs into the involved receiving data register, and update other registers to validate B_{RDR}. Whenever a receiving data register is read, we update relevant registers again to invalidate B_{RDR}, so that the software cannot read the data register when there is no data available. For WDR holes, we synthesize a function that sends the written data to the host, and update relavant registers to validate B_{WDR}. For HINT holes, we synthesize a function that evaluate and update the interrupt firing condition according to B_{HINT}. For each register involved in the condition, we insert a call to the introduced function after its value is updated to update the interrupt condition. For UNDR holes, we insert code after writes to related registers: if the written value meets the corresponding condition in B_{UNDR}, we update registers to reach the expected value as indicated in B_{UNDR}.

Complex Behaviors. The meanings of complex behaviors generated through on-demand analysis (see §4.3.2) are related with the semantics of the used commands, which are pre-defined by analysts according to the target peripheral. With such semantics, these behaviors can then be interpreted and utilized to complete the template. Note that for peripherals of the same kind, the involved commands as well as their semantics can be reused across different MCUs, and one only has to adapt the command contents for each different driver library family. Following this procedure, we successfully synthesized DMA models for STM32 MCUs with only 3 commands.

5 Evaluation

We implement PERRY for ARM Cortex-M MCUs using Clang/LLVM [33], KLEE [4] and Z3 [11], which consists of about 19,000 lines of C++ and Python code. In this section, we first present details of our testing environment (§5.1), then we organize the results of our study around four key research questions (RQ) we attempt to answer (§5.2). Those research questions are listed below:

(RQ 1): How effective is PERRY in inferring hardware behaviors from drivers, considering the adoption of symbolic execution (i.e., efficiency)?

(RQ 2): Are the inferred hardware behaviors consistent with actual hardware behaviors (i.e., consistency)?

(RQ 3): Can hardware models generated by PERRY be used to emulate various firmware (i.e., universality)?

(RQ 4): Can hardware models generated by PERRY be easily fixed or extended to support missing hardware functionalities (i.e., scalability)?

Table 2: Summary of selected libraries

Vendor	Driver Library	MCUs
ST	STM32CubeF0 v1.11.3	STM32F0 series
	STM32CubeF1 v1.8.4	STM32F1 series
	STM32CubeF4 v1.26.2	STM32F4 series
	STM32CubeF7 v1.16.2	STM32F7 series
	STM32CubeL0 v1.12.1	STM32L0 series
NXP	MCUXpresso SDK v2.12.0	FRDM-K22F
	MCUXpresso SDK v2.11.0	FRDM-K64F
	MCUXpresso SDK v2.8.0	FRDM-K82F
	MCUXpresso SDK v2.2.0	FRDM-KL25Z
	MCUXpresso SDK v2.12.0	LPC51U68
Microchip	Advanced Software Framework v3.52.0	SAM4L-EK SAM4E Xplained Pro SAM4S Xplained SAM E70 Xplained SAM V71 Xplained Ultra SAM3X8E

5.1 Experiment Setup and Methodology

Experiment Setup. We select 10 driver libraries from three top MCU vendors as shown in Table 2. A driver library may cover a series of MCUs, e.g., the STM32CubeF4 driver library can be used by all STM32F4xx MCUs such as STM32F469/479, STM32F407/417 and STM32F413/F423. Therefore, the selected driver libraries can cover over 30 MCUs. For each driver library, the corresponding SVD files are already contained within driver libraries, and we only need to provide RAM/ROM range information for the hardware metadata file before the evaluation, which requires a small amount of one-time effort (specifying CPU models, memory regions, etc.) and is a common procedure in existing methods [15,44]. Driver libraries for NXP MCUs are generated by the MCUXpresso SDK builder tool [41] provided by NXP and hence have different versions. The driver library from Microchip supports all MCUs, but we have to compile it for each MCU. All experiments are conducted on an Intel Xeon E5-2620 v2 @ 2.10GHz machine running Ubuntu 20.04 TLS, equipped with 64GB memory.

Methodology. We first evaluate time consumption of applying PERRY to synthesize hardware models from the selected driver libraries to answer **RQ 1**. Then, we evaluate the passing rate achieved by the generated models on P2IM unit tests [15] to answer **RQ 2**. To address **RQ 3**, we use the hardware models to emulate firmware samples, inspect whether they can perform expected functionalities, and evaluate the success rate. To answer **RQ 4**, we use the

Table 3: **PERRY model consistency.**

Peri.	Unit test	STM32F103			FRDM-K64F	ATSAM3X8E		Passing Rate
		Arduino	RIOT*	NUTTX	RIOT	Arduino	RIOT	
ADC	read converted values	✓	-	✓	✓	✓	✓	5/5
DAC	write values for conversion	-	-	-	-	✓	✓	2/2
GPIO	execute the interrupt callback	✓	✗(RCC ✧)	✓	✓	✓	✓	5/6
	read a pin	✓	✗(RCC ✧)	✓	✓	✓	✓	5/6
	set/clear a pin	✓	✗(RCC ✧)	✓	✓	✓	✓	5/6
PWM	perform basic configuration	✓	-	✓	✓	✓	✓	5/5
I2C	receive bytes	✗(▲)	-	✗(▲)	✗(✧)	✓	-	1/4
	send bytes	✗(▲)	-	-	✗(✧)	✓	-	1/3
UART	receive bytes	✓	✗(RCC ✧)	✓	✓	✗(✧)	✗(✧)	3/6
	transmit bytes	✓	✗(RCC ✧)	✓	✓	✗(✧)	✓	4/6
SPI	receive bytes	✓	✗(RCC ✧)	✓	✓	✓	✓	5/6
	transmit bytes	✓	✗(RCC ✧)	-	✓	✓	✓	4/5
TIMER	execute the interrupt callback	-	✗(RCC ✧)	-	✓	-	✓	2/3
	read counter values	-	✗(RCC ✧)	-	✓	-	✓	2/3
LoC to Fix		3 (1 for RCC, 2 for I2C)			1 (for I2C)	2 (for UART)		**49/66(74.24%)**

* All STM32F103/RIOT unit tests failed due to a single wrong behavior in the RCC peripheral. Unit tests marked with "-" do not exist. ✧ represents implicit assumptions on hardware and ▲ represents in-context register operations.

LoC required to fix/extend models as metric to evaluate the involved manual efforts.

5.2 Experiment Results

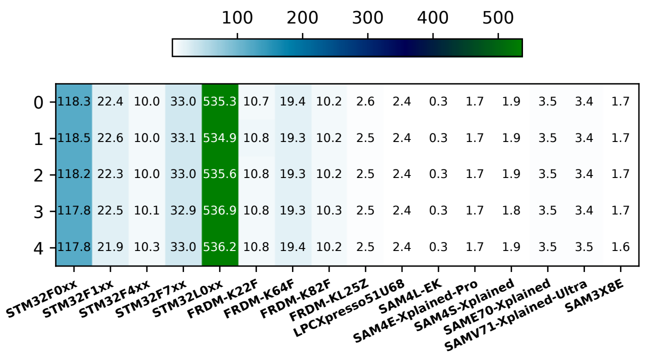

Figure 3: Model synthesis time consumption (# minutes).

Efficiency (*to Answer RQ1*). We use PERRY to synthesize peripheral models for all MCUs listed in Table 2. We conduct experiments for each MCU 5 times to eliminate the influence of random factors. We list the number of collected traces, the number of inferred hardware behaviors, and synthesized peripheral models in Table 4. We use time consumption to evaluate PERRY's efficiency in synthesizing hardware models, and the results are listed in Figure 3.

The results demonstrate that PERRY can effectively collect execution traces by symbolically exploring driver library paths and infer peripheral behaviors for most MCUs within reasonable time budget (i.e., less than 33 minutes). PERRY needs time (~9 hours) when the target driver library is over complicated (e.g., 4,820,407 traces from the STM32CubeL0 driver library).

Consistency (*to Answer RQ2*). We validate the consistency of the synthesized models by PERRY using P2IM unit tests [15]. The authors of P2IM provide 48 firmware images consisting of 66 valid unit tests to validate the functionality of emulated hardware platforms. These unit tests cover eight categories of peripherals on different combinations of three MCUs (STM32F103, FRDM-K64F and SAM3X8E) and three OS libraries (Arduino, RIOT and NuttX). We now compare PERRY with SEmu [62] because SEmu also aims to synthesize universal hardware models. We do not compare PERRY with path exploration oriented methods including Laelaps, P2IM, uEmu, and Fuzzware, because they only generate firmware-specific models as discussed in §2.2.

Table 3 shows the evaluation results. Without manual intervention, PERRY passes 49 (74.24%) unit tests, while SEmu fails all of them. SEmu cannot infer hardware behaviors for clock configuration peripherals (RCC, MCG and PMC) and the involved firmware samples cannot boot. SEmu fails to infer such behaviors because they are not described in the hardware manual. However, the driver source code presents such behaviors and PERRY successfully captures them. PERRY fails to pass 17 unit tests due to 6 wrong or missing hardware behaviors. After fixing or adding these behaviors with 6 LoC, PERRY achieves a passing rate of 100%. In comparison, SEmu achieves the same goal by manually fixing and adding 16 rules.

Universality (*to Answer RQ3*). We demonstrate the universality of the hardware models by emulating various firmware. Note that we reuse the models from the consistency evaluation. The used firmware samples consist of two parts: 10 real-world firmware samples from P2IM [15], and 19 shell firmware from two popular RTOS (LiteOS [25] and Zephyr [58]). A piece of firmware is successfully emulated if it can perform expected tasks defined by its code logic.

We collect the number of missing and wrong behaviors for

Table 4: PERRY model synthesis efficiency.

MCUs	# Traces	# HB	Peripherals*
STM32F0xx	7,915,522	60	RCC, SPI, USART, DMA, I2C, TIM
STM32F1xx	927,325	52	RCC, SPI, USART, I2C, DMA, TIM, ADC, EXTI, FLASH
STM32F4xx	231,159	52	PWR, RCC, RTC, SPI, USART, I2C, ADC, RNG, TIM
STM32F7xx	1,331,346	64	PWR, RCC, SPI, USART, I2C, RNG, TIM
STM32L0xx	4,820,407	64	DMA, I2C, RCC, RTC, SPI, TIM, USART
FRDM-K22F	68,598	36	I2C, MCG, SPI, USART
FRDM-K64F	278,881	41	MCG, SPI, USART, I2C, PORT
FRDM-K82F	20,908	38	MCG, SPI, LPUART, I2C
FRDM-KL25Z	22,168	40	MCG, SPI, I2C, UART0, UART1/2
LPC51U68	38,340	28	ASYNC_SYSCON, SPI, SYSCON, USART, I2C
SAM4L-EK	4,491	16	HCACHE, HFLASH, SCIF, USART, SPI
SAM4E Xplained Pro	6,203	36	PMC, UART, USART, PIO, SPI, TWI
SAM4S Xplained	6,257	41	PMC, UART, USART, PIO, SPI, TWI
SAM E70 Xplained	6,260	57	PMC, UART, USART, PIO, SPI
SAM V71 Xplained Ultra	6,193	55	PMC, UART, USART, PIO
SAM3X8E	2,796	32	PIO, PMC, SPI, TC, TWI, UART, USART

* Peripherals with no hardware behaviors extracted are omitted.
* **"HB"** represents hardware behavior.

each hardware model during emulation and the number of LoC to fix these models. The evaluation results are shown in Table 5. Without manual intervention, PERRY successfully emulates 20 (%68.97) firmware. After manually fixing problematic models (see scalability evaluation), we successfully emulate all firmware.

To demonstrate that these models can actually help the emulated firmware to perform meaningful tasks, we conduct two case studies on the CNC and Drone firmware. Without the synthesized hardware models, both firmware cannot even boot. This demonstrates that hardware models generated by PERRY are necessary and sufficient for firmware to perform meaningful tasks.

The CNC firmware includes a G-Code interpreter that controls the movement of the underlying hardware. The firmware accepts G-Code inputs from USART. Our hardware model directly connects the emulated USART hardware to `stdin` of the host. Every time we enter G-Code into `stdin`, the emulated USART is notified, certain status register bits of the hardware are set, and the incoming data is stored in the data register. Hence, the firmware can bypass checks of these status bits, retrieve the data within the data register, parse it, and perform predefined functionalities.

The Drone firmware reads sensor data through I2C, control the position of the drone accordingly, and report drone state using USART. The emulated USART and I2C are connected to separate pipes for data injection, and set certain status register bits upon receiving incoming data. We inject random inputs to the I2C as sensor data, and monitor the output from the USART. As a result, the firmware can continuously read sensor data and perform position control, and we can inspect the running state of the drone through USART output.

Table 5: PERRY model universality.

MCUs	Firmware	# Miss. Behaviors	# Wrong Behaviors	LoC to Fix
STM32F0 series	Zephyr-Shell LiteOS-Shell	1 (☆)	1 (▲)	4
STM32F1 series	Zephyr-Shell LiteOS-Shell Drone Gateway Reflow_Oven Robot Soldering_Iron	0	0	0
STM32F4 series	Zephyr-Shell LiteOS-Shell CNC PLC	0	1 (▲)	1
STM32F7 series	Zephyr-Shell LiteOS-Shell	0	1 (▲)	1
STM32L0 series	Zephyr-Shell LiteOS-Shell	1 (☆)	0	3
FRDM-K22F	Zephyr-Shell	0	0	0
FRDM-K64F	Zephyr-Shell Console	0	0	0
FRDM-K82F	Zephyr-Shell	0	0	0
FRDM-KL25Z	Zephyr-Shell	0	0	0
SAM4L-EK	Zephyr-Shell	2 (✧)	0	4
SAM4E Xplained Pro	Zephyr-Shell	2 (✧)	0	4
SAM4S Xplained	Zephyr-Shell	2 (✧)	0	4
SAM E70 Xplained	Zephyr-Shell	2 (✧)	0	4
SAM V71 Xplained Ultra	Zephyr-Shell	2 (✧)	0	4
SAM3X8E	Heat_Press Steering_Control	0	0	0

Note ☆: Non-trivial hardware functionalities. ✧: Implicit assumptions on Hardware. ▲: In-context register operations.

Scalability (*to Answer RQ4*). For encountered missing or wrong behaviors, we manually extend or fix them according to corresponding hardware manuals. By thoroughly investigating these failing cases, we summarize main causes as listed below.

- **Implicit Assumptions on Hardware:** Drivers assume the hardware always functions as expected. Therefore, drivers may perform certain operations on the hardware without checking every related registers. For example, drivers can write a register and implicitly assume that another register is updated automatically by hardware without waiting for it. As a consequence, such implicit behaviors cannot be captured by PERRY.
- **In-Context Register Operations:** A register might get updated after a sequence of register operations, but the UNDR pattern makes PERRY to falsely believe that only the last register operation leads to the update.
- **Non-trivial Hardware Functionalities:** Some firmware samples utilize non-trivial hardware functionalities (e.g.,

interrupt table relocation), which PERRY fails to synthesize behaviors.

We use the number of LoC to measure the required manual efforts, and each line contains only one instruction. The results are shown in Table 3 and Table 5. In total, we add or modify 35 LoC to fix generated hardware models, and it only takes at most 4 LoC to fix one model. Note that fixing models for MCUs of the same family (e.g., STM32F4 and STM32F7) even requires less manual efforts because their peripherals are highly similar, resulting in similar fixes. Additionally, as a benefit of our method, fixed models can be used to emulate other firmware without further modification, e.g., for MCUs involved in P2IM unit tests, PERRY achieves a 100% success rate when emulating firmware samples running on them. Therefore, the scalability of PERRY is demonstrated by the minimum manual efforts it requires.

6 Security Applications

As discussed in §3.1, bugs within drivers and applications endanger the firmware. Therefore, we conduct three case studies (**CS**) to demonstrate PERRY's capability in catching driver bugs, as well as reproducing and finding vulnerabilities in firmware application code. Note that we reuse hardware models from §5 without additional refinements.

6.1 Security Application in Drivers

(CS-I) Mining Specification Violation Bugs. PERRY infers hardware behaviors from drivers when generating hardware models, and we can compare these inferred hardware behaviors with hardware manuals to see whether the driver implementation follows the specification. We manually check the hardware behaviors inferred from STM32Cube and MCUXpresso drivers and identify two inconsistent bugs. We reported the two bugs to corresponding vendors, and they have acknowledged and fixed them. We demonstrate this process using the bug found in the RCC driver for STM32F0 MCUs. The hardware behavior is inferred from Listing 3, where the hardware is supposed to set the HSI48RDY bit of the CR2 register after the HSI48ON bit of CR2 is set by the driver, which, if taken literally, is correct.

```
1 #define __HAL_RCC_HSI48_ENABLE()  \
2     SET_BIT(RCC->CR2, RCC_CR2_HSI48ON)
3 __HAL_RCC_HSI48_ENABLE();
4 tickstart = HAL_GetTick();
5 while(__HAL_RCC_GET_FLAG(RCC_FLAG_HSI48RDY) == RESET) { ... }
```

Listing 3: STM32F0 RCC driver code snippet.

However, the macro RCC_FLAG_HSI48RDY is falsely defined to check the HSI48ON bit. As a result, the inferred hardware behavior indicates that it is the HSI48ON bit, instead of the HSI48RDY bit, that should be set by the hardware after it is set by the software, which conflicts with the specification. This bug will cause the firmware to directly perform subsequent operations without waiting the HSI14 oscillator to stabilize (because the condition at line 5 is always evaluated to false), which may lead to unexpected errors.

6.2 Security Application in Firmware Applications

(CS-II) Reproducing Firmware Vulnerabilities. In this case study, we use PERRY to reproduce two vulnerabilities (CVE-2022-1041 and CVE-2022-1042) in Zephyr's Bluetooth protocol stack. The two vulnerabilities are located in the BLE host [59, 60], and can be triggered by malformed Host Controller Interface (HCI) packets during BLE mesh provisioning. HCI packets are exchanged between BLE host and controller over USB or UART.

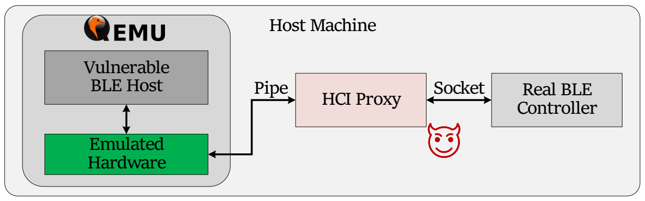

Figure 4: Reproducing BLE host vulnerabilities with PERRY.

Figure 4 depicts our vulnerability reproducing method. The vulnerable Zephyr BLE host firmware runs inside QEMU, using the STM32F4xx hardware model generated by PERRY. The emulated vulnerable BLE host exchanges HCI packets over the emulated UART peripheral with interrupt enabled, while the emulated UART is connected to a HCI packet proxy using pipes, and the proxy communicates with the BLE controller on the host machine though sockets. The proxy forwards HCI packets between the emulated BLE host and the real BLE controller as if they're directly communicating with each other. We inject malicious HCI packets into the emulated HCI host through the proxy to trigger the vulnerabilities. To validate the effectiveness of the payload, we place break points on bug triggering locations and inspect the program state using GDB. As a result, we successfully reproduce both vulnerabilities.

(CS-III) Fuzzing RTOS for Vulnerabilities. In this case study, we fuzz LiteOS by emulating it using PERRY-generated hardware models. The overall fuzzing approach is inspired by TriforceAFL [24]. We implement fuzz drivers for the MQTT and LWM2M protocol stack of LiteOS, and compile it against the STM32F769 Discovery MCU board. Additionally, we implement various sanitizers (UBSAN [7], KASAN [17], KMSAN [19] and KCSAN [18]) for LiteOS

Table 6: Results of a 6-hour fuzzing campaign on LiteOS.

Component	Target	Speed (#/sec)	# Exec.	# Path	# Vuln.
MQTT	`Deserialize_ack`	518.99	11,400,193	9	0
	`Deserialize_connack`	729.21	15,757,024	11	0
	`Deserialize_connect`	1233.25	26,547,383	35	1
	`Deserialize_publish`	634.81	13,632,707	14	1
	`Deserialize_suback`	517.32	11,380,177	13	1
	`Deserialize_subscribe`	969.45	20,505,021	12	1
	`Deserialize_unsuback`	469.20	10,261,209	10	0
	`Deserialize_unsubscribe`	537.79	11,910,801	11	1
LWM2M	`coap_parse_message`	331.79	7,331,274	4,112	3
	`lwm2m_data_parse(TLV)`	271.10	6,261,404	6,000	0
	`lwm2m_data_parse(JSON)`	10.84	1,638,116	3,160	2

to facilitate vulnerability detection. The target firmware is emulated using the STM32F7xx hardware model generated by PERRY. We also modify QEMU to collect coverage using dynamic binary instrumentation (DBI), such that a coverage-guided fuzzer (i.e., AFL [57]) can be integrated.

We develop 8 fuzz drivers for the MQTT protocol stack and 3 fuzz drivers for the LWM2M protocol stack. Fuzz drivers are compiled with KASAN and UBSAN enabled. We run each fuzz driver for 6 hours and the statistics are listed in Table 6. Through fuzzing, we totally uncover 10 vulnerabilities. 7 are new vulnerabilities and 3 have been reported by previous efforts [34], which were not fixed until we reported them again. The discovered vulnerabilities are all out-of-bound access vulnerabilities. We also reported newly discovered vulnerabilities to relevant parties. Three of them have been acknowledged and fixed with CVE-2021-41040 assigned. The remaining four vulnerabilities were directly fixed in LiteOS, as they were linked to a ported third-party library. Unfortunately, the developers of the library have not responded to us as of the time of writing.

7 Threats to Validity

In assessing the validity of our study, several potential threats emerge, particularly related to the limitations of PERRY in inferring hardware behaviors from drivers and the manual effort required to utilize PERRY effectively:

Limited Hardware Behaviors. Although PERRY enables on-demand analysis to infer complicated hardware behaviors, the categories of hardware behaviors that PERRY can infer from drivers are limited. For example, PERRY fails to infer non-trivial hardware behaviors like IVT relocation, due to their inconspicuous characteristics in drivers. However, these behaviors can be implemented based on the generated hardware models with little manual efforts.

Manual Efforts. Although PERRY substantially automates the process of synthesizing hardware models, it still demands manual involvement in three specific areas. Initially, an analyst must prepare the relevant hardware metadata for each driver library and utilize the provided compiler-wrapper to compile the library into LLVM bitcode. Secondly, the generated hardware models must be integrated into an emulator, which necessitates altering the emulator's build script. Lastly, despite these tasks being minimal, it is incumbent upon the analyst to manually correct any incorrect or missing hardware behaviors in the generated model.

8 Related Work

Firmware Rehosting and Analysis. Firmware can be analyzed statically to find bugs [10, 22, 42, 48], thus eliminating the requirement for firmware rehosting techniques. To take advantage of dynamic analysis techniques, efforts have been made to rehost firmware. Hardware-oriented approaches assume the presence of actual physical devices and can be divided into two categories: hardware-in-the-loop and record-and-replay. The first kind of approaches [9, 31, 56] only emulate CPUs and forward peripheral accesses to actual devices. The second kind of approaches [20, 49] record peripheral accesses during firmware executions and replay them when emulating firmware. The dependency on real devices has limited the scalability of these methods.

Table 7: Hardware-free MCU rehosting techniques.

Study	Input	Model Generation	Faithful Model	RQ2.Model Consistency	Firmware-independent	Exec. Low-Level Code	Full System
HALucinator [8]		✗			✗	✗	✓
BaseSAFE [39]		✗			✗	✗	✗
Para-rehosting [34]		✗			✗	✗	✓
P2IM [15]	Firmware	✓	✗		✗	✓	✓
Laelaps [5]	Firmware	✓	✗		✗	✓	✓
uEmu [61]	Firmware	✓	✗		✗	✓	✓
Fuzzware [44]	Firmware	✓	✗		✗	✓	✓
SEmu [62]	Manual	✓	✓	✗	✓	✓	✓
PERRY	Driver	✓	✓	✓	✓	✓	✓

Firmware-oriented approaches address this limitation by providing different virtual execution environments for different firmware, and can be performed at either register-level or function-level as listed in Table 7. Function-level approaches replace certain firmware functions to avoid hardware interactions [6, 8, 27, 29, 34, 39]. Register-level approaches infer how peripherals respond to register accesses, and are designed to explore firmware paths instead of faithfully emulating peripheral hardware [5, 15, 28, 37, 40, 44, 61]. These methods generate firmware-specific hardware models, which suffer from low universality and low fidelity. Function-level and register-level approaches can be combined [21, 23, 43] to emulate complex firmware like TrustZone OS.

A concurrent work [62] aims to mitigate the above limitations by generating hardware models with high fidelity under the guidance of hardware specifications. This method involves a considerable amount of manual labor, while our work is more automated.

Belief-based Software Analysis. Belief analysis [13] is a technique to analyze a program's assumptions on certain program properties, and has been successfully applied to discover bugs [3, 13] and infer specifications for programs [32]. Unlike these approaches, our work extend belief analysis to infer hardware models instead of software properties.

9 Conclusion

In this paper, we propose PERRY, a system that synthesizes faithful and extendable MCU peripheral models from peripheral drivers. Through a comprehensive evaluation, we demonstrate the efficiency, consistency, universality, and scalability of PERRY in synthesizing hardware models. With a 74.24% unit test passing rate, PERRY efficiently infers hardware behaviors and enables high-fidelity emulation of various firmware, while also allowing for easy extension with minimal manual efforts. Through our case studies, we showcase the security applications of PERRY in reproducing firmware vulnerabilities, detecting specification-violation bugs in drivers and fuzzing RTOS for new vulnerabilities. These results highlight the potential of PERRY in enhancing security-oriented tasks of IoT firmware analysis.

Acknowledgments

We thank the shepherd and the anonymous reviewers for their valuable suggestions and comments. This research was supported in part by National Natural Science Foundation of China Grant No. 62232004, by US National Science Foundation (NSF) Awards 1931871 and 2325451, by Jiangsu Provincial Key R&D Programs Grant Nos. BE2021729, BE2022680, and BE2022065-5, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant No. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Arm, "System view description," 2022. [Online]. Available: https://arm-software.github.io/CMSIS_5/SVD/html/index.html

[2] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC)*, 2005.

[3] F. Brown, A. Nötzli, and D. Engler, "How to build static checking systems using orders of magnitude less code," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[4] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[5] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[6] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[7] Clang, "Undefined Behavior Sanitizer documentation," 2023. [Online]. Available: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[8] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[9] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-Wide security testing of Real-World embedded systems software," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[10] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[11] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[12] R. Electronics, "Flexible software package," 2023. [Online]. Available: https://github.com/renesas/fsp

[13] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[14] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory, D. Balzarotti, and W. Robertson, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2021.

[15] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[16] D. Giese, "Having fun with iot: Reverse engineering and hacking of xiaomi iot devices," 2018. [Online]. Available: https://dontvacuum.me/talks/DEFCON26/DEFCON26-Having_fun_with_IoT-Xiaomi.pdf

[17] Google, "Kernel address sanitizer," 2023. [Online]. Available: https://github.com/google/kernel-sanitizers

[18] ——, "Kernel concurrency sanitizer," 2023. [Online]. Available: https://github.com/google/kernel-sanitizers

[19] ——, "Kernel memory sanitizer," 2023. [Online]. Available: https://github.com/google/kernel-sanitizers

[20] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[21] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of Real-World TrustZone software using emulation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[22] G. Hernandez, F. Fowze, D. J. Tian, T. Yavuz, and K. R. B. Butler, "FirmUSB: Vetting USB device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[23] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, and K. R. B. Butler, "Firmwire: Transparent dynamic analysis for cellular baseband firmware," in *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS)*, 2022.

[24] J. Hertz and T. Newsham, "Triforceafl," 2017. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[25] Huawei, "Huawei LiteOS," 2023. [Online]. Available: https://gitee.com/LiteOS/LiteOS

[26] T. Instruments, "TI MCU+ SDK," 2023. [Online]. Available: https://github.com/TexasInstruments/mcupsdk-core

[27] M. Jiang, L. Ma, Y. Zhou, Q. Liu, C. Zhang, Z. Wang, X. Luo, L. Wu, and K. Ren, "ECMO: peripheral transplantation to rehost embedded linux kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[28] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[29] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards large-scale emulation of iot firmware for dynamic analysis," in *In Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[30] D. Komaromy and L. Szabo, "How to tame your unicorn - Exploring and exploiting zero-click remote interfaces of modern huawei smartphones," in *Proceedings of the 2021 BlackHat USA (BHUSA)*, 2021.

[31] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling Near-Real-Time dynamic analyses of embedded systems," in *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[32] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[33] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, 2004.

[34] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From library portability to para-rehosting: Natively executing opensource microcontroller oss on commodity hardware," in *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS)*, 2021.

[35] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022.

[36] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, "Security vulnerabilities of internet of things: A case study of the smart plug system," *IEEE Internet of Things Journal (IoT-J)*, vol. 4, no. 6, pp. 1899–1909, 2017.

[37] Q. Liu, C. Zhang, L. Ma, M. Jiang, Y. Zhou, L. Wu, W. Shen, X. Luo, Y. Liu, and K. Ren, "Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[38] K. Lu and H. Hu, "Where does it go?: Refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[39] D. Maier, L. Seidel, and S. Park, "BaseSAFE: baseband sanitized fuzzing through emulation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2020.

[40] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of dma input channels for dynamic firmware analysis," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, 2020.

[41] NXP, "MCUXpresso SDK Builder," 2023. [Online]. Available: https://mcuxpresso.nxp.com/en/welcome

[42] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "KARONTE: Detecting insecure multi-binary interactions in embedded firmware," in *In Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[43] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[44] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*, 2022.

[45] G. Semiconductor, "Gd32f4xx firmware library," 2023. [Online]. Available: https://www.gd32mcu.com/data/documents/toolSoftware/GD32F4xx_Firmware_Library_V3.1.0.7z

[46] N. Semiconductor, "Standalone drivers for peripherals present in nordic socs," 2023. [Online]. Available: https://github.com/NordicSemiconductor/nrfx

[47] N. Semiconductors, "Mcuxpresso SDK," 2023. [Online]. Available: https://github.com/nxp-mcuxpresso/mcux-sdk

[48] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[49] C. Spensky, A. Machiry, N. Redini, C. Unger, G. Foster, E. Blasband, H. Okhravi, C. Kruegel, and G. Vigna, "Conware: Automated modeling of hardware peripherals," in *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2021.

[50] STMicroelectronics, "RM0385 reference manual," 2018. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0385-stm32f75xxx-and-stm32f74xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

[51] ——, "Stm32cubef7 MCU firmware package," 2023. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF7

[52] E. Systems, "Espressif iot development framework," 2023. [Online]. Available: https://github.com/espressif/esp-idf

[53] I. Technologies, "Psoc 6 peripheral driver library," 2023. [Online]. Available: https://github.com/Infineon/psoc6pdl

[54] M. Technology, "Advanced software framework," 2023. [Online]. Available: https://www.microchip.com/en-us/tools-resources/develop/libraries/advanced-software-framework

[55] N. Technology, "M460 series cmsis bsp," 2023. [Online]. Available: https://github.com/OpenNuvoton/m460bsp

[56] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[57] M. Zalewski, "American fuzzy lop," 2018. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[58] Zephyr, "Zephyr project," 2023. [Online]. Available: https://www.zephyrproject.org/

[59] Y. Zhang and Z. Lin, "When good becomes evil: Tracking bluetooth low energy devices via allowlist-based side channel and its countermeasure," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[60] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking secure pairing of bluetooth low energy using downgrade attacks," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[61] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[62] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, "What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

A Write and Update Semantic Inference

A.1 Bit Value Variation-Based Write Semantic Inference

```
1 #define __HAL_RCC_HSI14_ENABLE() (RCC->CR2 |= RCC_CR2_HSI14ON)
2 /* Update CR2 to Enable HSI14 */
3 __HAL_RCC_HSI14_ENABLE();
4 ...
5 /* Wait till HSI14RDY is set in CR2 */
6 while(__HAL_RCC_GET_FLAG(RCC_FLAG_HSI14RDY) == RESET) {
7   if((HAL_GetTick() - tickstart) > HSI14_TIMEOUT_VALUE) {
8     return HAL_TIMEOUT;
9 }}
```

Listing 4: STM32F0xx RCC driver code snippet.

To infer write semantics in UNDR beliefs, considering all possible concrete values that can be written into Reg_A is impractical. However, we have observed that UNDR beliefs typically involve bit-level granularity, where only specific bits, known as critical bits, are intended to be toggled. For example, in Listing 4 at line 3, the critical bit is HSI14ON. Leveraging this observation, we propose a bit value variation-based method to identify critical bits and infer write semantics. The approach compares the values of each bit in the target register before and after the write operation. If a bit's value changes after the write, we consider the new value as the write semantic.

Formally, we define three states for bit values: MUST_ONE, MUST_ZERO, and ANY. These states indicate whether a bit must be 1, must be 0, or can have any value, respectively. To analyze the bit values, we utilize the bit-blasting technique to obtain symbolic expressions for each bit in both the original register value and the written value. Then, using an SMT solver, we determine the ranges of possible bit values for each bit. We formulate queries to the SMT solver based on the path constraints CS, asking whether a bit can be 1 (query Q_1) or 0 (query Q_0). The resulting bit state ST is determined as follows:

$$ST = \begin{cases} \texttt{MUST\_ONE}, if(Q_0, Q_1) = (unsat, sat) \\ \texttt{MUST\_ZERO}, if(Q_0, Q_1) = (sat, unsat) \\ \texttt{ANY}, else \end{cases} \quad (5)$$

By diffing bit states after and before the write, critical bits can be identified using the following function

$$f\left(ST_b^i, ST_a^i\right) = \left(ST_b^i \neq ST_a^i\right) \wedge \left(ST_a^i \neq \texttt{ANY}\right) \quad (6)$$

where ST_b^i and ST_a^i denotes the bit status of the i-th bit before and after the write. The concrete value v for each identified critical bit b can be calculated with the $eval$ function:

$$eval\,(\texttt{MUST\_ONE}) = 1, eval\,(\texttt{MUST\_ZERO}) = 0 \quad (7)$$

For each identified critical bit b and its concrete value v identified, we introduce a constraint $BC_b := (b = v)$. Since the update semantics can be expressed as the access constraints C of the register read access, the hardware behavior can be derived as:

$$B_{UNDR} := \left(\bigwedge_b BC_b, C\right) \quad (8)$$

A.2 Write-Update Dependency Resolution

In the example listed in Listing 5, the value bitMask is written to the PRESETCTRLSET register, and the PRESETCTRL register is expected to get updated to bitMask. However, since bitMask is not constrained, the write and update semantics cannot be inferred using previous methods.

```
/* set bit */
SYSCON->PRESETCTRLSET[regIndex] = bitMask;
/* wait until it reads 0b1 */
while (0u == (SYSCON->PRESETCTRL[regIndex] & bitMask)) {}
```

Listing 5: LPC51U68 SYSCON driver code snippet.

We observe that the expected update value for Reg_B is typically linearly related to the value written into Reg_A. To address this, we propose synthesizing linear formulas that satisfy these relationships. The approach involves establishing a linear equation $val_u = A \times val_w + B$, where val_u represents the expected value for Reg_B, val_w represents the value written into Reg_A, and A and B are constants to be determined. By substituting Reg_B in the access constraints C_B, we use a SMT solver to find concrete values for A and B that ensure the constraints in C_B are always true. In the given example, the substituted C_B becomes $\texttt{(A} \times \texttt{bitMask + B) \& bitMask} \neq \texttt{0}$. With the determined values for A and B, we successfully synthesize a linear formula between val_u and val_w, allowing us to derive the hardware behavior as:

$$B_{UNDR} := (Reg_A = val_w, Reg_B = A \times val_w + B) \quad (9)$$