What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation

Wei Zhou*†
School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, Hubei, China
NCNIPC, UCAS
Beijing, China

Lan Zhang*
College of Information Sciences and Technology, Penn State University
State College, PA, USA

Le Guan
School of Computing, University of Georgia
Athens, GA, USA

Peng Liu
College of Information Sciences and Technology, Penn State University
State College, PA, USA

Yuqing Zhang†
NCNIPC, UCAS
Beijing, China

ABSTRACT

Emulating firmware of microcontrollers is challenging due to the lack of peripheral models. Existing work finds out how to respond to peripheral read operations by analyzing the target firmware. This is problematic because the firmware sometimes does not contain enough clues to support the emulation or even contains misleading information (*e.g.,* a buggy firmware). In this work, we propose a new approach that builds peripheral models from the peripheral specification. Using NLP, we translate peripheral behaviors documented in chip manuals into a set of structured condition-action rules. By checking, executing, and chaining them at run time, we can dynamically synthesize a peripheral model for each firmware execution. The extracted condition-action rules might not be complete or even be wrong. We, therefore, propose incorporating symbolic execution to assist in pinpointing the root cause and correcting the problematic rules. We have implemented our idea for five popular MCU boards. Using a new edit-distance-based algorithm to calculate trace differences, our evaluation against a large firmware corpus confirmed that our prototype achieves much higher fidelity compared with state-of-the-art solutions. Benefiting from the accurate emulation, our emulator effectively avoids false positives observed in existing fuzzing work. We also designed a new dynamic analysis method to perform driver code compliance checks against the specification. We found some non-compliance which we later confirmed to be bugs caused by race conditions.

*Both authors contributed equally to this research.
†Corresponding authors: Wei Zhou, Yuqing Zhang.
The extended version of the paper with more details can be found at https://arxiv.org/abs/2208.07833.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559386

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

firmware emulation; microcontroller; NLP; fuzzing

ACM Reference Format:
Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. 2022. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation. In *Proceedings of Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3559386

1 INTRODUCTION

Microcontroller units (MCUs) are small, resource-constraint SoCs (system on a chip) that drive a number of security- and safety-critical application fields such as smart homes, mobile robots, and healthcare. With the emergence of Internet of Things (IoT) technology, security analysis (*e.g.,* bug finding and taint analysis) of MCU-based IoT devices is becoming increasingly important. Due to the tight coupling between firmware code and physical peripherals, IoT devices are usually tested as a whole in the real world. However, involving real peripherals can make it significantly more difficult to achieve scalably (*e.g.,* scale to thousands of vendors and millions of peripherals) security analysis of firmware and devices. Because of this, in recent years, several emulation-based techniques have been proposed [3, 5, 12, 16, 19, 35, 39, 49]. By rehosting the firmware on a PC, many existing dynamic security analysis methods become possible without involving real hardware.

Problems. A primary challenge in firmware emulation is modeling the behaviors of unknown peripherals. In particular, when the firmware reads a peripheral register, how can we generate an appropriate response? A common philosophy adopted by existing work is that *the emulator should generate responses that meet the expectations of firmware so that it does not crash or hang.* In this sense, they try to learn knowledge (about unknown peripherals) from the firmware itself. We call this line of work *firmware-guided* solutions. For example, P^{2}IM [12] automatically builds peripheral

models by observing the peripheral access patterns when exercising the target firmware in the emulator. Laelaps [3], *μEmu* [49], Jetset [19], and Fuzzware [35] use symbolic execution to explore the firmware to find satisfactory response values. However, the firmware sometimes contains *incomplete* or even *misleading* information. For the formal, take the timing of interrupts as an example. Existing emulators have no clue regarding which interrupt should be delivered at a particular time. More generally, existing methods only learn coarse-grained static peripheral models from the firmware. We will shortly illustrate the missing information (not contained in firmware) and the negative effects with concrete examples (see Section 3.1). For the latter, if the firmware itself contains a memory bug, existing emulators such as *μEmu* [49] would try to avoid executing code containing it, which is a wrong emulation. As another example, P^{2}IM reports many mis-categorizations of registers due to the misleading access patterns, which ultimately leads to incorrect peripheral models [12].

The incomplete and misleading information leads to serious low fidelity issues, influencing the effectiveness, efficiency, and applicability of firmware analysis tools that are built on top of the emulator. **1) Effectiveness:** Driver code for complex peripherals cannot be properly emulated. This would exclude a large amount of real-world firmware in dynamic analyses. For example, none of existing work can fuzz firmware with the Ethernet functionality. **2) Efficiency:** Even if the target firmware can be emulated, the execution trace may deviate significantly from that on real hardware. This introduces non-negligible false positives (*i.e.,* find non-existing bugs on infeasible paths) and false negatives (*i.e.,* fail to find real bugs due to missing code coverage). Moreover, failing to deliver interrupts accurately makes emulation progress slow (see Section 3.1). **3) Applicability:** Low-fidelity emulation also limits the types of dynamic analysis that can be adopted. In fact, we observe that all the related work on firmware emulation is limited to fuzz testing [3, 12, 19, 35, 49] since fuzzing can naturally tolerate inaccurate emulation — humans have to manually confirm the bugs anyway. A high-fidelity emulator can push forward with new security analysis applications other than fuzzing.

A New Direction in Automated Firmware Emulation. Different from existing solutions which are guided by the knowledge inferred from the target firmware, we propose *specification-guided* emulation. The specification guides the emulator in a principled way, making emulation more accurate. Consequently, this approach can emulate more complex firmware, run dynamic analysis more efficiently, and be applied to more accurate analysis tasks (*e.g.,* specification-based compliance check in our prototype).

Main Observations. To specify the behaviors of a peripheral circuit, state diagrams and state tables are the *de facto* standard [37]. In fact, we frequently observe the use of finite state machines to implement both peripheral drivers in firmware and peripheral backends in the emulator. Unfortunately, other than during the internal design phase, there is no widespread adoption of any state-diagram-based peripheral specification which can be directly interpreted by formal methods [10]. On the contrary, MCU chip vendors typically release a text-based reference manual for each chip, in which a natural language (in particular English) is used to describe peripheral behaviors. Compared with formatted language or tables used to describe a state diagram, the language used in chip manuals are more ad-hoc. For example, we found many sentences similar to "if the value of register *reg_A* is equal to *X*, then the register *reg_B* will be assigned with a value *Y*". On the one hand, these ad-hoc sentences *equivalently* describe the same peripheral behaviors as a state diagram does. In fact, it is a common practice for developers to read and comprehend the manuals, and then construct state machines to implement drivers or emulator backends. On the other hand, it is extremely hard, if not impossible to *automatically* composite these ad-hoc sentences into a state diagram.

This work does not aim at the ambitious goal of automatically constructing a state machine to implement peripheral backends in emulators. However, we have the following observations that help us more accurately model peripherals using specifications. 1) Individual sentences in chip manuals can be easily understood by modern natural language processing (NLP) engines and be formalized as some simple **condition-action** (C-A) rules; 2) Without explicitly maintaining a state machine, strategically executing these C-A rules can also emulate peripherals.

Proposed Solution. Based on the aforementioned observations, we propose to use peripheral specification documented in chip manuals to model its behavior. The resulting model targets a piece of hardware instead of the firmware. This will fundamentally address the issues associated with existing firmware-guided emulation solutions as mentioned before (*i.e.,* incomplete and misleading information contained in the firmware).

After obtaining enough C-A rules, at run time, we dynamically synthesize a model for each peripheral. Each **peripheral model** provides appropriate responses on behalf of the corresponding hardware. In the meanwhile, we maintain the status of a peripheral by selectively executing certain relevant C-A rules. Based on our empirical study, executing these rules implicitly maintains the corresponding state machine. In Section 3.2, we show an intuitive example. The maintained status will in turn help the model select the right C-A rules when processing the next firmware request.

Due to poor documentation of the manuals and limitations of the NLP engine itself, we occasionally observe missing or wrong C-A rules. To address this problem, we leveraged symbolic execution with invalid state detection to quickly diagnose the root cause. Intuitively, if a C-A rule is missing or wrong, it is likely that emulation would enter an invalid state [49]. If this happens, we use symbolic execution to identify which peripheral read operation is responsible for the error and correspondingly fix the C-A rule. It is worth noting that our NLP-based approach and invalidity-guided emulation are complementary to each other. If there exists a missing/faulty C-A rule, invalidity-guided emulation can help find it. But invalidity-guided emulation such as *μEmu* [49] alone cannot address the "incomplete and misleading information contained in firmware" issue for the reasons we mentioned before. Concretely, *μEmu* does not know when to issue an interrupt and non-crashing execution cannot guarantee data correctness.

We have implemented the proposed idea with a prototype named *SEmu* (Specification-guided EMUlator), and evaluated it systematically. *SEmu* automatically extracted C-A rules from five chip manuals for STM32F1 [41], STM32F4 [42], STM32L1 [40], NXP K64 series [32], and Atmel SMART series [27]. We also diagnosed and enhanced these results with the help of invalidity-guided emulation [49]. Collectively, these peripheral models allow us to test

and evaluate the same set of firmware samples used in existing work [12, 16, 49]. To objectively compare our solution with existing work, we developed a new method based on edit distance to quantify emulation fidelity, which measures how an emulated execution deviates from real execution on hardware. Using it, we conclude that our approach achieves much better trace fidelity compared with existing firmware-guided solutions. This allows us to emulate more complex peripheral and get better fuzzing results. Finally, we applied our approach to a new dynamic analysis task which we call *specification-based compliance check*. It checks whether peripheral driver implementations comply with the information extracted from chip manuals. We found some non-compliance which was later confirmed to be programming bugs. We released our artifacts at https://github.com/MCUSec/SEmu.

To summarize, we have made the following contributes:

- We proposed specification-guided firmware emulation to more accurately emulate firmware. The core technique is to leverage NLP techniques to automatically extract useful C-A rules from chip manuals.
- We proposed incorporating invalidity-guided emulation to identify missing or faulty C-A rules extracted by *SEmu*.
- To quantitatively evaluate emulation fidelity, we proposed a new method based on modified edit distance.
- We implemented our idea and compared the peripheral models generated by *SEmu* against manually constructed ones and automatically constructed ones.
- The extracted peripheral behavior rules enabled us to conduct driver code compliance check, and we have uncovered real non-compliance bugs with it.

2 BACKGROUND

2.1 Overview of MCU Peripherals

The main functions of MCU devices are accomplished by controlling the peripherals to interact with external environments. Therefore, the firmware running on the MCU can be considered as a collection of peripheral usages. To properly operate a peripheral, the firmware should make sure that the peripheral is in the right status before any access; otherwise, unexpected behavior would occur. For example, if the peripheral is busying doing hardware operations (*e.g.,* an analog-to-digital converter, or ADC, is converting the external analogue values to the digital values), it cannot respond to firmware functions. To communicate such important status information (*i.e.,* whether the peripheral is ready to respond to a certain access request), the firmware typically first inquires about the current peripheral status by reading the corresponding status register (`SR`), which is memory mapped into the address space of processor (a.k.a. MMIO). Peripherals also notify the processor (thus the firmware) of the external events via the interrupt mechanism. Typically, an external event causes a state change in the peripheral. In addition, some high-throughput peripherals like I2C, USB, and Ethernet may use Direct Memory Access (DMA) to transfer the data between the RAM and peripheral without involving the CPU.

2.2 MCU Reference Manuals

As can be seen, even a simple peripheral involves lots of hardware states. To help firmware developers understand peripheral behaviors, chip vendors commonly publish reference manuals for each chip written using natural language. MCU reference manual is supposed to provide essential information on how to use every peripheral, including its registers, memory map, hardware interfaces, and behaviors. Since each peripheral contains multiple registers, the manual at least holds a section of **register memory map** to summarize the MMIO address for each peripheral register and its access permission (*e.g.,* read only, write only and read/write). Following the memory map, the functions of each register are explicated in addition to a **field description** table. The field description table specifies the meaning of individual fields in the registers, including their names, bits, access permissions, and functions. In explaining the function of each field, it first describes how the value of the field will change. Then, it enumerates the possible values and explains the corresponding meanings. Taking the UART peripheral in an NXP **K64F** as an example, we found the following sentence to describe the `RDRF` field of the `SR1` register:

> RDRF is set when the number of datawords in the receive buffer is equal to or more than the number indicated by RWFIFO[RXWATER]. 1

This sentence specifies how the field `RDRF` will be changed to 1, which we refer to as a condition. Based on the function, a register field can be categorized into one of the following types. A **status field** is used to indicate the current status of a peripheral. A **control field** is used by the firmware to configure or initialize the peripheral functions. A **data field** serves the I/O interface of the peripheral. It is usually connected with a shift register which is further backed by a data buffer. While some peripherals use the same data register to connect both the transmit buffer (for sending data) and the receive buffer (for receiving data), some use two dedicated data registers for sending and receiving data.

The manual also includes an **interrupt vector assignment** table that summarizes the IRQ numbers assigned for each peripheral. Note some peripherals may have multiple IRQ numbers for different functions. For example, a peripheral may generate an interrupt on error, alarm, or update. Finally, each DMA request of a peripheral is mapped into a DMA channel. This information is summarized in a table called **DMA channel assignment**. For example, the DMA request from the ADC1 peripheral of the F103 device is routed to channel 1, which is activated by programming the DMA control bit of the corresponding peripheral. When a DMA transfer ends, the corresponding interrupt is triggered to notify the firmware.

As mentioned before, chip vendors commonly publish reference manuals for developers, and our proposed approach relies on the availability of these manuals and the important information included in them (*i.e.,* register memory map, field description, etc.). To show the prevalence of chip manuals (thus the applicability of our approach), we conducted a survey for 15 top players in the global MCU market [26]. We found that all major chip vendors publish chip manuals and the manuals contain the information mentioned above for many peripherals. It is worth mentioning that different vendors may use different names for semantically similar sections. For example, the "register memory map section" is called "memory map and register boundary addresses" in STM32 manuals [42]. On the other hand, a chip manual may not include comprehensive information for all peripherals. For example, in the manual of a popular BLE chip [30] designed by Nordic, developers

are redirected to read the BlueTooth Core Specification to fully understand its BLE peripherals. Our approach cannot model these peripherals.

2.3 Natural Language Processing

Although reference manuals (mainly in PDF format) are unstructured data, there are observable characteristics in the format and text. Hence, NLP techniques can be used to understand the naturally expressed sentences and extract condition-action logic from chip manuals. More specifically, NLP techniques can identify the related registers and fields by recognizing the noun and verb phrases, find the causal relationships between the conditions and actions through the sentence structure and conjunctions, and infer the semantics of the sentences based on the relation between pairs of words.

We reuse the quoted sentence above as an example. It has three named entities: `RDRF`, `receive buffer`, and `RWFIFO[RXWATER]`. The *Part-of-speech (POS) tagging* technique [25] can be applied to recognize the named entities by identifying the characteristic structure of words (*e.g.,* noun and verb). The causality can be distilled through *Constituency Analysis* [50]. A binary parse tree is generated to divide a sentence into different constituents so that the conditions and actions can be separated. In this example, the identified action is "RDRF is set" and the remaining sub-sentence is a conditional clause. Also, *Typed Dependencies Analysis* [4] can be applied to analyze the grammatical structure. It matches the verbs and their corresponding subjects or objects, so that the `RDRF` sentence can be converted to first-order logic. Taking the sub-sentence "RDRF is set" as an example, `set` is a predicate that depends on the noun phrase `RDRF`. Since this kind of condition-action representations are repeatedly found in chip manuals, NLP techniques can be very effective in extracting C-A rules.

3 MOTIVATION AND KEY IDEAS

We first explain how existing low-fidelity emulation solutions can negatively impact fuzz testing. Then, we use an intuitive example to show how our approach can achieve higher fidelity.

3.1 Problems with Low-fidelity Emulation

To facilitate large-scale dynamic firmware analysis, full-system emulation without any hardware dependence is essential. Recent work has made substantial progress in peripheral modeling, a key barrier to dynamic firmware analysis [3, 5, 12, 16, 19, 35, 39, 49]. Different techniques have been explored recently, such as access-pattern recognition [12], symbolic execution [3, 19, 35, 49], and real trace analysis [16, 39]. All these solutions share the same philosophy — as long as the peripheral model meets the expectations of the firmware, the emulation is considered successful. Here, meeting the expectations means the firmware does not crash or hang. Therefore, they only use the firmware as the source of information to build coarse-grained static peripheral models. These firmware-guided solutions only approximately emulate the firmware and thus suffer from low fidelity issues when the input space becomes large.

In Listing 1, we show a code snippet from a real-world firmware sample. It runs on a PLC (Programmable Logic Controller) that communicates with a supervisory machine via the Modbus protocol over UART. The main program logic is implemented by the `loop()` function that is invoked in each scan cycle. This function first checks whether there is any incoming data available (line 3). Only when more than seven bytes are accumulated in the buffer (line 4) will they be fetched and processed (line 6). To fill the receive buffer, the UART interrupt must be raised by hardware so that the corresponding handler (`UART_IRQHandler()`) will be invoked. The handler not only receives data, but also sends data and handles errors. The exact sub-function (in the handler function) to invoke is jointly decided by the status register (*i.e.,* `isrflags`) and two control registers (*i.e.,* `crlflags` and `cr3flags`).

Before presenting the problem, we first explain how the UART hardware works together with the shown driver code in receiving external data. Whenever a byte is available, the hardware moves a byte to a FIFO receive buffer inside the UART hardware, and then sets the `RXNE` flag of the UART status register. If the interrupt is enabled (*i.e.,* the flag `RXNEIE` is set), an interrupt will also be triggered. According to current status and control registers values, the handler will eventually execute the `UART_Receive_IT()` function which reads the data register. On reading the data register, the FIFO receive buffer is shifted so that a byte is returned via the data register to the firmware.

In this example, the approximate emulation achieved in existing firmware-guided work exhibits at least two defects in fuzzing, which are fundamentally caused by the incomplete information contained in the firmware. First, the emulator does not know the type and timing of interrupt deliveries. Therefore, it triggers each active interrupt in a round-robin fashion. However, there are many active interrupts during firmware execution. It typically takes lots of time for the UART interrupt to be selected by the emulator. Second, even if the UART interrupt is triggered, the status and control registers should hold correct values so that the right sub-function (in our example, `UART_Receive_IT()`) can be invoked. However, existing work cannot guarantee this since other sub-functions do not crash the execution either and thus can also be selected. Obviously, the firmware cannot tell the emulator which sub-function should be executed simply because the relevant information is not contained in the firmware. Combined, it usually takes a long time or relies on some non-determinisms of fuzzing to invoke the intended function.

To conclude, firmware lacks some essential information for the firmware-guided emulators. Therefore, these emulators have to blindly try all possible combinations in the input space (*e.g.,* interrupt timing and status register values). While these solutions can avoid crashes and hangs due to some clever designs, they cannot achieve high fidelity.

```
1 int __fastcall loop(Modbus *const buffer, ...) {
2     ...
3     int length = HardwareSerial::available();
4     if (length <= 7)
5         return 0;
6     Modbus::getRxBuffer(buffer);
7     ...
8 }
9
10 void UART_IRQHandler(UART_Handle *hUART) {
11     uint32_t isrflags = READ_REG(hUART->SR);
12     uint32_t cr1flags = READ_REG(hUART->CR1);
13     uint32_t cr3flags = READ_REG(hUART->CR3);
14     uint32_t errorflags = (isrflags & ...)
15     /* UART in Receiver mode */
16     if((isrflags & UART_SR_RXNE) != RESET && ...){
17         UART_Receive_IT(hUART);
18         ...
19     }
20     /* UART in Transmitter mode */
```

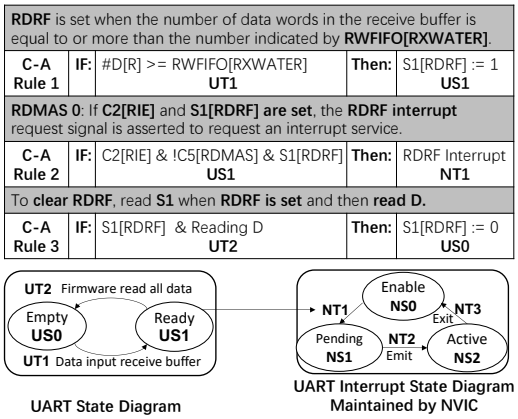


Figure 1: Checking, executing, and chaining condition-action rules can emulate state transitions (In C-A rule 2, "RDMAS 0" indicates the rule is valid only when the RDMAS field is 0).

```
21    if((isrflags & UART_SR_TXE)) && ...){
22        UART_Transmit_IT(hUART);
23        ...
24    }
25    errorflags = (isrflags & (UART_SR_PE | UART_SR_FE |
        UART_SR_ORE | UART_SR_NE));
26    /* UART in Error mode */
27    if((errorflags != RESET) && ...)
28        ...
29    ...
30 }
```

Listing 1: Code snippet of real-world PLC firmware (minor modifications were made to save space).

3.2 Emulating State Transitions with C-A Rules

To address the low-fidelity problem, our approach automatically extracts structured C-A rules from chip manuals and strategically executes them. This process emulates the state transitions of the underlying hardware without *explicitly* maintaining the state machine, making the emulation more principled and thus yielding good fidelity. We use Figure 1 to illustrate the idea using the UART peripheral as an example. The top table lists three sentences excerpted from the reference manual for NXP K64F series chips [32] and C-A rules extracted from them. We highlight the **named entities** that serve as the subjects/objects in these sentences. At the bottom, we show the state diagrams that we manually constructed by reading the manual. C-A rule 1 states that if the condition "#D[R] >= RWFIFO[RXWATER]" is satisfied (UT1), then an action "S1[RDRF] := 1" will be invoked (US1). Note how this C-A rule corresponds to the edge UT1 and node US1 of the UART state diagram. The action is to assign value 1 to the register field `S1[RDRF]`, which in turn makes the condition of C-A rule 2 true (US1). A UART interrupt is generated as a result of the corresponding action (NT1), making the UART interrupt pending (NS1). Note that this action has a cross-peripheral effect that influences the state machine of NVIC, Arm's built-in interrupt management peripheral. The C-A rule 3 is related to receiving data. How it maps the state transitions is self-explanatory. As can be seen, by simply checking, executing, and chaining these C-A rules, we can emulate the same state transitions governed by the manually constructed state diagrams.

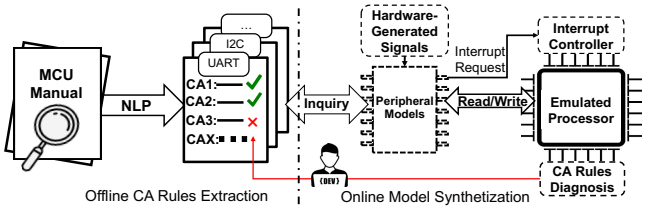


Figure 2: *SEmu* overview

4 OVERVIEW

We leverage peripheral specification to guide the emulator to overcome the aforementioned limitations of previous work. In a sense, we mimic a real-world emulator development process in which the developers read the chip manuals and then write peripheral models as the emulator backends based on the knowledge from the manuals. However, to make it scalable, we adopted a semi-automatic method by leveraging the latest advances in NLP. As shown in Figure 2, given a chip manual, we use an NLP engine to extract a set of **Condition-Action (C-A)** rules that describe peripheral behaviors (Section 5). The key observation is that although the chip manuals are diverse, most sentences are similarly structured to reduce ambiguity, making it possible to accurately extract C-A rules. With these rules, at run time, the emulator intercepts the firmware-peripheral interactions, which drive the checking, executing and chaining of C-A rules. As mentioned before, this emulates state transitions of peripheral hardware without explicitly maintaining the state machine. When firmware issues a read request to a peripheral register, the emulator inquires the current peripheral status and calculates a response. Therefore, our approach dynamically builds a peripheral model according to the specification, achieving *state-aware* firmware emulation (Section 6).

Due to imperfect documentation and limitation of the NLP engine itself, we occasionally observed failed emulation. Therefore, we also developed a symbolic-execution-aided diagnosis technique to quickly pinpoint the faulty or missing C-A rules. The diagnosis results are manually analyzed by human analysts who then revise the C-A rules to fix the problem (Section 7).

On top of the proposed state-aware emulator, we developed two security analysis plugins. First, an AFL-based fuzzer is developed to find memory-related bugs in firmware. Second, we develop a compliance check tool that tracks the MMIO access sequence for each peripheral and compares it with the static rules in specification. Unless otherwise specified, we use peripheral descriptions from the NXP K64F manual [32] as examples to illustrate ideas.

5 EXTRACTING C-A RULES USING NLP

In this section, we first introduce the formal definition of conditions and actions, along with their classifications. Then, we elaborate on how to identify conditional clauses and extract trigger-action rules.

5.1 Conditions and Actions

The condition-action rule is the most critical concept in our system. It describes how important events such as accessing an MMIO register influence the peripheral state. A *condition* comprises one or more predicates combined using the Boolean *and* operator. An *action* is one or more assignment functions. A C-A rule connects one

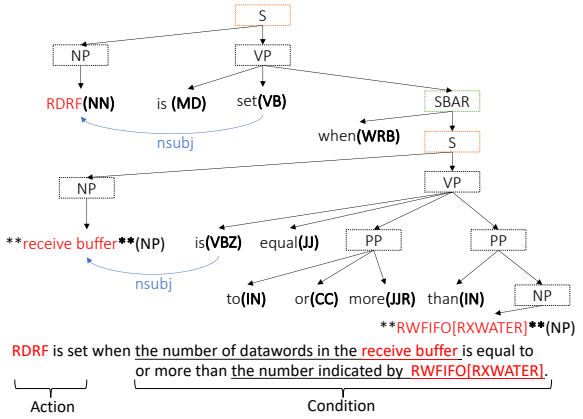
condition and one action – when the condition is met, the paired action will be taken. If multiple conditions are satisfied, all the associated actions should be executed. The subjects/objects in a C-A rule are modeled as named entities (*e.g.,* a register field). Therefore, each predicate in a condition specifies whether the value of a named entity is equal to, greater than, or less than a reference value or the value of another named entity. Each assignment function assigns a value to a named entity. Formally, a C-A rule is represented as the following logical expression: $P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow F_1, F_2, ..., F_m$, where P refers to a predicate and F refers to an assignment function.

Conditions are event-driven. That is, when certain events happen, a condition might be satisfied. We group conditions based on the relevant named entities and firmware operations. Events are delivered via signals. Based on how the underlying signal is generated, we classify the affected conditions into three types. **Type-1 condition (via external-hardware-generated signals):** These signals are driven by external hardware events, such as receiving new data from the physical UART interface. The result can often be abstracted as filling an internal buffer with new data. The description for this type of conditions usually contains a keyword "hardware" or "buffer". In the quoted sentence of Section 2.2, "the number of data words in receive buffer" which is the subject of the condition, is related to a hardware-generated signal. **Type-2 condition (via firmware-generated signals)**: This type of conditions is driven by firmware operations. For example, in the sentence "LBKDIF is cleared by writing a 1 to it", the write operation is a signal to update the value of `LBKDIF`. **Type-3 condition (via internal signals)**: When a register value has been updated by due to the execution of any previous actions, some related conditions may become true. We call this chained C-A rule execution. Therefore, each relevant C-A rules are checked when a previous action causes a register update. We frequently found interrupt delivery or DMA request as a result of chained C-A rule execution. The C-A rule 2 in Figure 1 shows such an example.

Depending on how the firmware interacts with the peripheral (*i.e.,* MMIO access and interrupts/DMA requests), there are three types of actions. **Type-1 action (MMIO register related):** These actions update the value of a register field. In the previous example, the action is "RDRF is set", which sets 1 to the `RDRF` bit of register `S1`. **Type-2 action (interrupt related):** These actions send an interrupt request to the interrupt controller. **Type-3 action (DMA related):** These actions generate a DMA transfer request or an interrupt request when a DMA transfer is completed. These two types of actions are normally triggered by buffer-related signals. For example, the sentence describing C-A rule 2 in Figure 1 elucidates a C-A rule containing a type-2 action; meanwhile, when the `RDMAS` field holds 1 as indicated in the manual, the sentence, "RDMAS 1: if C2[RIE] and S1[RDRF] are set, the RDRF DMA request signal is asserted to request a DMA transfer" will generate a type-3 action.

5.2 Workflow of C-A Rule Extraction

There are several challenges to extract the relevant C-A rules automatically: 1) How can we identify the sentences which are associated with C-A rules? 2) How can we recognize and handle co-references which are very common? 3) How can we identify the causal relationship of the sentences? These challenges are addressed with the following main steps.



1. Bold characters are the parts of speech (POS) [25] (*e.g.,* NN for nouns).
2. Blue edges label the dependencies between two words. (*e.g.,* `nsubj` indicates the dependency between a predicate and a subject).
3. Red characters are named entities.
4. **Phrase** is in abbreviated form. The full phrase is underlined in the sentence.

Figure 3: Constituency analysis of the example sentence of Section 2.2.

5.2.1 Collecting Relevant Sentences via Named Entity. A chip manual contains thousands of sentences, many of which are not related to any meaningful C-A rules and should be filtered out. To collect relevant sentences, we take advantage of named entities. As mentioned before, the subjects/objects in C-A rules are represented as named entities. We first identify a set of important named entities. Based on them, we directly match the occurrences in a sentence to collect relevant sentences. We consider two sources of named entities. First, firmware interacts with peripherals via the MMIO registers. Therefore, peripheral registers and their fields are a major source. Second, we also consider the receive/transmit buffers which are connected with data registers as named entities.

The initial set of named entities is extracted from the **register memory map** section of the manual. This is straightforward because this section is the central place where registers are enumerated. Then we scan the **fields description** part to find relevant sentences that contain at least one named entity. During this process, we extend the set of named entities with newly encountered subjects/objects, such as Ethernet transmit descriptors (TDES0). With the new entities, we also extend the search scope to other sections in the manual such as *functional description* and *application information*. As new named entities and relevant sentences are found, we will re-run the search algorithm over the already scanned sections. This is an iterative process which ends when no new named entities or sentences can be found, or exceeding a threshold (three by default in our prototype).

5.2.2 Identifying Coreferences. A well-known challenge in NLP is that multiple expressions in the natural language can refer to the same entity. For example, we have encountered at least the following different expressions for the UART status register: status register of UART, `UARTx_SR`, and `SR`. To uniformly represent them in C-A rules, we have to identify all these co-references. We address this issue by matching the noun phrases with the initial set of named entities extracted from the **register memory map**, because noun phrases in a sentence are often related to named

entities. Noun phrases are marked as "NP" in *Stanford POS Tagger* [25]. For instance, the underlined phrase "the number indicated by RWFIFO[RXWATER]" in Figure 3 is a noun phrase. We use approximate string matching to measure the similarity between an unknown noun phrase with each of the initial set of named entities to identify possible co-references. In this example, the word "RWFIFO[RXWATER]" is close to "UART FIFO Receive Watermark(UARTx_RWFIFO)". Thus, we know "RWFIFO" is a synonym of the register with the named entity "UARTx_RWFIFO". By further searching the field description of the register `UARTx_RWFIFO`, we identify "RXWATER" as a field of this register.

5.2.3 Identifying Conditions and Actions. Given a sentence, we then apply *Stanford Constituency Parser* [50], a popular language model, to analyze the grammatical structures of sentences. The conditional clauses (in the sentence) can be identified through constituency analysis as shown in Figure 3 which uses the quoted sentence in Section 2.2 as an example. Here, a sentence "S" is defined as a noun phrase "NP" followed by a verb phrase "VP". The sub-tree with "SBAR" as the root node stands for a subordinate conditional clause starting with the conjunction "When". After identifying the conditional clause, the sentence is divided into two sub-sentences indicating the condition and the action, respectively. Due to the complexity of natural language, sometimes the Stanford parser cannot identify complicated conditional clauses. For example, the Stanford parser cannot recognize the conditional clause in the sentence "LBKDIF is cleared by writing a 1 to it", To address this issue, we extend the grammar patterns used by the parser to specify syntactic constituents. Grammar patterns are a series of regular expressions that divide a sentence into several clauses. In the above example, two sub-sentences, $< NP, VBZ, VBN >$ and $< VBG, NP, IN, NP >$, are matched. Here, $VB.*$ represents different verb forms, such as gerund/present participle (VBG), present tense (VBZ), and past participle(VBN). IN matches all preposition/subordinating conjunctions (*e.g.,* in, of, to). The second sub-sentence "writing a 1 to it" is a prepositional phrase for the first sub-sentence "LBKDIF is cleared". So, we assign the second sub-sentence as the condition and the first one as the action.

5.2.4 C-A Rule Representation. After fully "understanding" the sentences, our NLP engine outputs a formal representation of C-A rules for the online model synthetization module. We first translate conditions and actions into predicates and assignment functions. To do so, we employ the *Stanford Dependency Parser* [4] to connect a named entity with the related verbs. As shown in Figure 3, the verb "set" is connected to the named entity `RDRF`. Then, the value and operator semantics involved in the verb phrase need to be extracted. For this purpose, we build a mapping table, in which one or more keywords are mapped to a particular binary value (*e.g.,* keyword "set" is mapped to the value 1 and "cleared" is mapped to `0`)) or Boolean operators (*e.g.,* the Boolean operator "equal to or more than" is mapped to `>=` and the assignment function is mapped to `:=`). Note the construction of the mapping table requires domain knowledge but is a one-time effort.

Second, we formulate rule triggers and actions. Rule triggers are recognized by parsing condition descriptions. For instance, for the sentence "the `LBKDIF` field is cleared when firmware writes 1 to it", we should check the corresponding C-A rule only when the firmware writes to this field. We define five types of triggers. 1) `B-triggers` represent the conditions which are checked when data in receive/transmit buffer has changed; 2) `W-triggers` represent the conditions which are checked on firmware write operations; 3) `R-triggers` represent the conditions which are checked on firmware read operations; 4) `V-triggers` represent the conditions which are checked when the value of a field is updated by internal signals; 5) `O-triggers` represent the conditions which are checked on any other signals such as the timer or manual invocations. If one rule comprises multiple conditions, we use the `&` symbol to concatenate them. It is straightforward to recognize actions from the parsed sentences. They are encoded depending on their types mentioned in Section 5.1.

Lastly, we formulate the C-A rules. The named entities are formally represented as `Reg[Field]` (*e.g.,* `RWFIFO[RXWATER]`). For data registers, we use `D[R]` and `D[T]` to represent the corresponding receiver and transmit buffers. `#` at the beginning of each buffer-related keyword indicates the current occupied size of that buffer. We use the $\rightarrow$ symbol to separate the conditions and actions in a C-A rule. Formal representation of the resulting C-A rules can be found in the extended version of the paper. We use the first sentence in Figure 1 to exemplify how a sentence is formulated. The translated C-A rule is:

$$B \quad \#D[R] \geq RWFIFO[RXWATER] \rightarrow S1[RDRF] := 1$$

"B" indicates that this C-A rule is triggered by a B-trigger. The phrase "the number of datawords in the receive buffer" is formalized as `#D[R]`, which should be "equal to or more than" RWFIFO[RXWATER]. The latter entity is directly extracted from the sentence and it already meets the required `Reg[Field]` format. The action is to assign 1 to "RDRF", which our parser automatically finds its holder register "S1", yielding an action "S1[RDRF] := 1".

6 SYNTHESIZING PERIPHERAL MODELS

We use QEMU to emulate the basic ARM ISA and core peripherals (*e.g.,* NVIC). During firmware emulation, we dynamically build a model for each peripheral, taking the intercepted firmware-peripheral interactions and the extracted C-A rules as inputs. By capturing the rule triggers mentioned in Section 5.2.4, we check whether the condition of any C-A rule becomes satisfied. Since the intercepted interactions only contain the raw addresses, we have to first translate the named entities into target addresses, which is trivial because the **register memory map** contains the needed information. If the named entity matches a predicate and the condition is satisfied, the corresponding action will be executed. We initialize a data structure of each peripheral, which contains 1) the current values for the affiliated named entities, and 2) the states of interrupt and DMA request (if any).

As mentioned in Section 5.1, there are three types of conditions. They may become satisfied in the presence of triggers from external hardware, firmware interaction, or internal hardware. Among them, the firmware interactions can be directly captured by intercepting the MMIO accesses. Internal hardware triggers come from the results of previous C-A executions. Specifically, the outcome of a C-A rule execution may trigger another C-A rule execution. In essence, this enables **chained execution** of C-A rules, a very important aspect in emulating peripheral behaviors.

To capture triggers from *external* hardware, we observe that most hardware signals are related to data transmission. This allows

us to model hardware signals by monitoring the I/O interface, in particular the transmit and receive buffers. For example, the RDRF field in UART is set when the receive buffer is full. Since the hardware receive/transmit buffers are used for external I/O channel, we emulate these buffers with two byte arrays. When the firmware writes a byte to the data register, we move it to the transmit buffer. When the firmware read from the data register, we return a byte from the receive buffer like real hardware. Consequently, buffer-related conditions like the number of available bytes in a receive or transmit buffer can be easily emulated. Likewise, we can also emulate timer-based hardware signals with software. For the rest of few hardware signals which cannot be emulated, we keep the corresponding conditions to be always satisfied. This makes the associated actions get a fair chance to be invoked. For example, the SR[STRT] field of the ADC peripheral in F103 is set by hardware when regular channel conversion starts. Since we have no clue as to when the conversion would start, we set the SR[STRT] field to be always one. While this may influence emulation fidelity, we found it helpful in pushing emulation forward.

7 DIAGNOSING FAULTY/MISSING C-A RULES

The extracted C-A rules may be incomplete or incorrect due to two reasons. First, some hardware signals cannot be emulated. We use a workaround that sets the corresponding condition to be always true, leading to the incorrect execution of certain C-A rules. Second, state-of-the-art NLP techniques face difficulty in handling complex sentences with underlying or nested conditions. It would involve substantial human efforts to locate the root cause.

To reduce the needed manual efforts, we propose an invalid-state-detection-based C-A rule checker to automatically diagnose the faulty/missing C-A rules. Inspired by *μEmu* [49], we find symbolic execution very good at reasoning about the root cause of failed emulation. Failed emulation typically leads to an invalid execution state (*e.g.,* stall or crash), and symbolic execution can help us quickly locate the improper response for the peripheral reading. Concretely, for the target peripheral, we first prepare a firmware sample and a valid testcase that correctly executes with the firmware. This can be verified on real devices. Then we run this sample on *SEmu* and collect the concrete responses generated by the synthesized peripheral models. These concrete responses are fed to a symbolic execution engine and used to guide symbolic execution in selecting branches, similar to the concolic mode implemented in S2E [9]. If an invalid state is detected, it must be due to wrong C-A rules, because we have specifically selected a firmware sample and testcase that do not trigger any invalid state. Since an invalid state is encountered, one of previous responses from our peripheral model must be wrong. We will take the other branch in the last conditional statement and solve the corresponding symbolic variable. This symbolic variable tells us the address of the wrongly modeled register. We further compare the wrong value generated by our model with the one solved by symbolic execution engine to identify the incorrect bits. Finally, our tool lists all the executed C-A rules related to this field in reverse order. Humans must now be involved to confirm the wrong rule. Diagnosis is an iterative process. In each round, it fixes one C-A rule until the used firmware can be emulated correctly.

Taking the description of SR[MOSCXTS] for the PMC peripheral of the SAM3X chip as an example, our NLP engine failed to generate any C-A rule for this field at first. So our model used the reset value (zero) by default. However, during PMC initialization, the diagnosis tool detected an invalid state due to a wrong response from reading the SR register. In particular, our diagnosis tool indicated that the response should be 0x1 at that point. Checking the manual, it does not clearly state that this field depends on a hardware signal (*i.e.,* no "hardware" keyword in the description) and no conditional clause was found. We addressed this issue by adding a static rule that SR[MOSCXTS] should always be set.

8 EVALUATION

We have implemented the proposed ideas in a prototype called *SEmu*. To evaluate it, we aim to answer the following research questions. Q1) Can our NLP engine automatically extract C-A rules to describe peripheral behaviors? Q2) Can the diagnosis tool help correct incorrect rules? Q3) Are the extracted C-A rules complete and sound? Q4) Can peripheral models dynamically built with C-A rules provide higher fidelity compared with firmware-guided approaches? Q5) Will the improved emulation fidelity provide better performance? Q6) Can compliance check which requires more accurate peripheral modeling help us find bugs?

We selected five chip manuals that cover more than twenty popular MCU series including STM32**F103** [41], STM32**F429** [42], STM32**L152** [40], NXP **K64F** series [32] and Atmel **SAM3X** series [27] belonging to three top MCU vendors. It is worth noting that a manual can cover a series of MCU chips that share similar peripherals. For example, STM32**F429** [42] also covers STM32F405/415, STM32F407/417, STM32F427/437 and STM32F439 series MCUs. Supporting these MCU models allows us to test and evaluate the same set of firmware samples used in existing work [12, 16, 49]. We first extracted initial C-A rules from these manuals and evaluated if they can support emulating a set of unit-test samples (Section 8.1). We then conducted C-A rule diagnoses to improve the C-A rules and used the enhanced rules to test the same unit-tests and a set of real-world firmware and complex demo programs shipped with chip SDKs which cannot be handled in previous work (Section 8.2). To explain why the enhanced C-A rules include enough information for building peripheral models, we evaluated their faithfulness to the "ground-truth" used in QEMU peripheral models (Section 8.3). Using the proposed method to measure trace similarity, we further conducted emulation fidelity tests (Section 8.4), and fuzzed real-world firmware samples (Section 8.5). Finally, we performed the compliance check between the driver implementation and the specification (Section 8.6). All the experiments were conducted on a 16-core Intel Xeon Silver 4110 CPU @ 2.10GHz server with 48 GB DRAM running a Ubuntu 18.04 OS.

8.1 C-A Rule Extraction

For each manual, we extracted C-A rules for 26 popular peripherals, including ADC, I2C, SPI, GPIO, UART, Ethernet, etc. For each peripheral, we counted the number of involved sentences (**#Sent.**), registers and fields (**#Reg(Field)**). For the extracted C-A rules, we broke them down based on their categorization information mentioned in Section 5.1. We also counted the number of total rules and the number of revised rules if any. In total, we extracted 3,602 unique C-A rules from 23,424 sentences involving 26 different peripherals. The detailed statistics of raw C-A rules obtained via our NLP engine can be found in the extended version of the paper.

More than half of the conditions are triggered by firmware-generated signals (C2). This indicates that MMIO interaction contributes the most to peripheral operations. 15.8% of conditions are associated with hardware-generated signals (C1), including the I/O interactions and other hardware signals. The rest (20.9%) are triggered by internal conditions (C3). Most of them are related to interrupt or DMA requests that become active as a result of another C-A rule execution (*i.e.,* chained execution). For actions, 84.2% of C-A rules hold type-1 action (A1) to update the state of fields. More than 14.2% of C-A rules are designed to generate interrupts (A2) and 1.57% of rules are used to send DMA requests (A3).

In general, complex peripherals need more C-A rules, as exemplified by the Ethernet and PWM peripherals. The same type of peripheral on different MCU SoCs may exhibit different statistics, due to the different implementation and presentation flavors. For example, on SAM3X, the state information about interrupts, access permission and channel modes are separately maintained in three registers, while K64F and STM32 combine them in one register. Moreover, the number of C-A rules for GPIO on SAM3X is noticeably higher than others. We found that this is because the GPIO lines on SAM3X are managed by the PIO controller, which requires complex configuration via 32 programmable input/output lines with three registers.

Using the Raw C-A Rules for Firmware Evaluation. We used the raw C-A rules to build peripheral models for 66 unit-test samples released in the P^{2}IM paper [12]. These unit-tests run on three chips, covering various peripherals and OS libraries. For these 66 test cases[1], P^{2}IM and *µEmu* achieve 83% and 95% passing rates, respectively. Unfortunately, during firmware booting, our emulator failed to correctly run the clock configuration for these unit tests. Specifically, the raw C-A rules for RCC on STM32F103, MCG on K64 and PMC on SAM3X cannot faithfully build usable models for these peripherals. Since these peripherals provide clock sources that are on the non-bypassable booting bath, the models built from raw C-A rules failed all the unit tests. However, regarding individual peripherals, the raw C-A rules work correctly for most other peripherals without enhancement. After revising clock related rules, our method achieved a passing rate of 96.97%. We explain how we semi-automatically revise these rules in Section 8.2.

8.2 C-A Rule Enhancement

When using the raw C-A rules to synthesize peripheral models, we encountered failed emulations. This lies in the limitation of NLP techniques and unavoidable ambiguous or even wrong specification representations. For example, we observed some typos and reported them to relevant maintainers. While we cannot fundamentally solve this problem, a symbolic-execution-aided diagnostic tool has been developed to quickly find out which rule is incorrect. Following the descriptions mentioned in Section 7, we first prepared 61 firmware samples that demonstrate individual peripheral usages in different working modes (*e.g.,* polling, interrupt and DMA mode for UART) from vendor SDKs. We then ran the samples with *SEmu* and used the collected concrete responses to guide symbolic execution. If the symbolic execution engine enters an invalid state, it terminates and outputs diagnosis information including the involved registers and C-A rules. Note that when an invalid state is detected, it must be due to wrong C-A rules because we have made sure the tested samples and test-cases do not crash on real devices. In total, we added 26 rules (0.6%) and fixed 21 rules (0.5%). To be specific, we added 5 C-A rules to I2C (F103, F429 and L152, 15 rules in total), 3 rules to ADC DMA mode (SAM3X), 3 rules for Ethernet (F429), 4 rules for MCG (K64), and 1 rule for PMC (SMA3); we also modified 6 rules for RCC (F103, F429 and L152, 18 rules in total) and remove 3 useless rules for SPI debug mode (K64). More details are provided in the extended version of the paper.

We use RCC, the clock peripheral for STM32F103 to explain how incorrect rules were corrected with the help of our diagnostic tool. The original sentence to describe the `CFGR[SWS]` field of RCC is "set and cleared by hardware to indicate which clock source is used as system clock". Since the NLP engine does not know how hardware will set/clear the field, our tool generated a rule $\underline{O \quad * \rightarrow CFGR[SWS] := 0/1/2/3}$, which indicates that 1) this is an O-trigger (Section 5.2.4), 2) the condition is always satisfied, and 3) the action is to set `CFGR[SWS]` with a random value from 0-3 (since this is a 2-bit field). When we used this rule against the firmware sample, we found that the RCC driver always failed booting. Using the diagnostic tool, we were able to capture the invalid state and pinpoint this field. In particular, the concrete response for `CFGR[SWS]` generated by *SEmu* led the symbolic execution into an infinite loop. Tracing back, a symbolic value corresponding to `CFGR[SWS]` was detected. Then, we read the relevant sentences that describe `CFGR[SWS]` in the manual and found that `CFGR[SWS]` should follow the value in `CFGR[SW]`. Moreover, 3 is an impossible value. Therefore, we updated this rule as:

$$V \quad CFGR[SW] == 0 \rightarrow CFGR[SWS] := 0 \tag{1}$$

$$V \quad CFGR[SW] == 1 \rightarrow CFGR[SWS] := 1 \tag{2}$$

$$V \quad CFGR[SW] == 2 \rightarrow CFGR[SWS] := 2 \tag{3}$$

The condition is any update operations to `CFGR[SW]` and the action is to assign the same value to `CFGR[SWS]`.

Using the Enhanced C-A Rules for Firmware Evaluation. With the enhanced rules, especially those related to clocks, we reran the 66 unit-test samples from P^{2}IM. *SEmu* achieves a passing rate of 100%. We additionally collected 17 new samples that represent real-world applications or more complex demo programs with multiple peripherals and diverse working modes such as DMA. For example, the PinLock firmware runs on a smart lock, which reads a PIN number through a UART interface, hashes it to compare with a known hash, and sends a signal to unlock a digital lock if the PIN is correct. *SEmu* successfully emulated all of them, and neither P^{2}IM nor *µEmu* can run any of these samples, with one exception that *µEmu* succeeded in emulating `Shell`. In Section 8.4, we clearly show that *SEmu* achieves much better emulation fidelity.

8.3 Faithfulness of C-A Rules

We evaluated how faithful the C-A rules used in *SEmu* are to the chip manuals. To collect the ground truth, we can manually check all the sentences for a peripheral; however, this would be a tedious task that depends on the individual's perceptions. To mitigate possible bias, we instead used the working peripheral models that are already implemented by QEMU developers. These backend models are written in the C programming language and only implement essential peripheral logic for correctly emulating firmware. This

[1] This data set included 70 unit tests originally, but four were removed in an erratum [1].

Table 1: C-A Rule Faithfulness Compared with Rules Used in QEMU Models (STM32F405)

Peripheral	SLOC	# Common Rules	# Missing Rules in *SEmu*	# Missing Rule in QEMU
UART	145	10	0	16
SPI	123	5	0	13
ADC	200	9	2	46
Timer	221	1	0	28
EXTI	96	23	0	29
SYSCFG	83	7	0	1

avoids the distraction from massive amount of irrelevant sentences in the original manual. With the C source code for peripheral models, we then manually checked the program logic and translated them into C-A rules using the same format as described. If a C statement can be translated to a C-A rule, we commented this line with the rule. Taking the UART backend emulation for STM32F405 as an example [34], at line 67, when QEMU receives any data via the data register, the SR[RXNE] field is updated. Such logic can be translated into the C-A rule: $\frac{B \quad \#DR[R] \geq 0}{\rightarrow SR[RXNE] := 1}$. Then, we can compare the C-A rules generated by *SEmu* with the "ground-truth" derived from QEMU.

Following this method, we collected "ground-truth" rules for all QEMU-supported peripherals of STM32F405/205, including SYSCFG, UART, ADC, SPI, Timer and EXTI, from the source code of the QEMU release (ver. 6.1). Note *SEmu* supports 20 peripherals for the same chip. In Table 1, we list SLOC (source lines of code) for QEMU model implementations, the number of common (and consistent) rules, the number of missing rules in *SEmu* but present in QEMU, and the number of missing rules in QEMU but present in *SEmu*. In counting SLOC, we excluded code that does not contain peripheral logic, such as data structure definitions.

There is a substantial overlap between the two rule sets, which represents the most important peripheral logic. Our method failed to find two C-A rules for the ADC peripheral that are present in QEMU. As an example, in the ADC backend emulation for stm32f405/205, at line 75 [33], QEMU truncates the received data based on the value of `ADC_CR1[RES]`. However, in the original manual, it only states that `ADC_CR1[RES]` is used to select the resolution width of the conversion. No information is given regarding how `ADC_CR1[RES]` impacts truncation precision, which cannot be automatically inferred by *SEmu*. Fortunately, the wrongly truncated data does not influence the emulation capability of *SEmu*. That is why our diagnostic tool did not find this imperfection.

On the other hand, there are lots of missing rules in QEMU. The main reason is that QEMU fails to implement many non-standard peripheral functions. To "support" a peripheral, QEMU only needs to implement emulation for the most common logic of this peripheral. As such, we frequently observed comments similar to "[**] is not implemented, the registers are included for compability" in the QEMU source code. For example, QEMU currently does not support IRQ or DMA requests for ADC and SPI. As another example, based on the manual, the SPI peripheral provides two main functions, supporting either the SPI protocol or the I2S audio protocol. However, QEMU does not support any I2S functions. We acknowledge that many of the rules extracted by *SEmu* correspond to rarely used peripheral functions and were never executed in our evaluation. However, we did obverse some critical peripheral logic being overlooked by QEMU. For example, based on the STM32F4XX manual, once the conversion of the selected regular ADC channel is completed, the `EOC` (end of conversion) flag in the SR register should be set. However, we did not find such logic in QEMU source code. To confirm this problem, we ran the ADC demo from the official STM32F4XX SDK package [45] with the latest QEMU and the emulation hung waiting for `EOC` to be set.

8.4 Emulation Fidelity Test

We used unit-test samples from P^{2}IM to evaluate emulation fidelity, and compared the fidelity achieved by *SEmu* with those on *μEmu* and P^{2}IM, since these samples are mostly supported by all the related work. To get the reference data, we leveraged the external debug probes (*e.g.,* ST-Link [43] and OpenSDA [31]) to collect traces on real hardware. Unfortunately, Atmel SAM3X is not equipped with external debugging capability. Therefore, we only performed fidelity tests against firmware samples for NXP FRDM-K64F [15] and STMicroelectronics Nucleo-F103RB [44].

In the following paragraphs, we first introduce how to collect traces on real devices and emulators. Then, we explain a numeric metric to quantify the similarity between two execution traces. It addresses the non-determinism issue happening during firmware execution (*e.g.,* the same input on the same hardware can generate different traces). Lastly, we discuss the results.

8.4.1 Trace Collection. To collect traces on real devices, we used OpenOCD [18] and an external debugging dongle to connect the target boards to the remote *gdbserver* provided by the chip vendors. Using the debugger, we recorded the program counter of each instruction execution. Collecting execution traces of emulators was much easier. We directly logged the starting address of each to-be-executed translation block. Then, we aligned it with the disassembled firmware code to reconstruct the execution trace following the same format as that on the real device.

8.4.2 Metrics. To quantitatively measure emulation fidelity, we propose a new metric to indicate the similarity between execution traces. In particular, we use the execution trace on the real device as the reference. By representing the traces as a sequence of addresses, we can use traditional string distance algorithms such as Levenshtein distance (a.k.a edit distance) [29]. However, due to the non-determinism of firmware execution (*e.g.,* interrupt timing), directly using edit distance cannot reliably measure the true similarity. For example, with the same input and firmware, two executions on real device could yield very different traces, but both of them achieve 100% fidelity.

To address this issue, we divided each trace into three parts based on their functions: 1) The **initialization trace** covers the initialization functions like peripheral configurations. It ends at the start of the main function. 2) The **main loop trace** includes the main firmware business. In the unit-tests, it corresponds to the data transmission logic. We modified the unit-tests so that the main loop was executed five times. 3) The **interrupt trace** records the instructions executed in the interrupt context. We separate it from others to eliminate non-determinism.

For each part of trace, we use edit distance to measure the similarity. In computational linguistics and computer science, edit distance is used to quantify how dissimilar two strings are, specifically, what is the minimum operations (*i.e.,* deletion, insertion and substitution) needed to transfer one to another. Similarly, we can use this method to measure the distance from traces on an emulator to traces on

the real device. To be specific, the whole trace is presented as a sequence of addresses, which correspond to the starts of each basic block. Compared with the trace of real device, a deletion operation means the emulator mistakenly takes an additional basic block; an insertion operation means the emulator misses a basic block execution; a substitution operation means the emulator executes a different basic block. The fewer operations are needed, the more similar are the two traces. In the original algorithm, deletion, insertion and substitution weigh equally. However, in our case, deleting or inserting repeated sequences are considered less important. Indeed, missing some code execution or executing some wrong code usually have worse effects on firmware analysis compared with repeating previous traces. For example, firmware usually runs in a polling mode that waits for a certain value of the status register. Whether it runs 100 loops or 1,000 loops does not effectively influence firmware analysis. Therefore, we assign the weight of repeated deletion or insertion operations to 1 while the others to 2.

The trace distance between the emulator and real device is calculated by $D_{Emulator} = D_{init} + D_{main} + D_{irq}$, where D_{init}, D_{main}, and D_{irq} are the modified edit distances for the three parts of trace mentioned before. Finally, we normalize the measured distances into a number ranging from 0 to 1 to represent the fidelity score, with 1 being the most similar (*i.e.,* short distance). Concretely, the fidelity score of an emulator can be calculated by $1 - \min(\frac{D_{Emulator}}{D_{QEMU}}, 1)$, where D_{QEMU} is the edit distance measured on unmodified QEMU. Here, D_{QEMU} represents the worst case emulation result because no peripheral emulation is provided. As can be seen, if $D_{Emulator}$ is low enough, the fidelity score will be 1. If $D_{Emulator}$ is equal to or higher than D_{QEMU}, the fidelity score will be 0. Otherwise, the fidelity score ranges from 0 to 1.

8.4.3 Results. We measured fidelity scores for the three parts of the traces for I2C, UART, GPIO and Timer unit test samples. Under the proposed fidelity metric, *SEmu* is the only one that has perfect scores among the test samples. Due to the space limit, more details can be found in the extended version of the paper.

The most significant difference occurs in the interrupt trace, where both P^{2}IM and *μEmu* perform very badly. After manual analysis, we found that P^{2}IM mis-categorizes many control or status registers as data registers, which triggers unexpected emulation results. For example, when the transmit enable field (`CR[TE]`) is overwritten by external data, it could prevent firmware from executing the data transmission function. *μEmu*, on the other hand, suffers from low fidelity issues due to its unsound invalid state heuristics. For example, we found it frequently invoked infeasible paths which do not actually cause a hang or crash. Finally, neither P^{2}IM nor *μEmu* is aware of the interrupt timing. Therefore, the interrupt handler typically takes irrelevant paths.

8.5 Fuzz Testing

In this section, we evaluate *SEmu* regarding fuzzing. The tested firmware includes 10 samples used in P^{2}IM [11] and 2 complex samples used in Pretender [17]. We also added 4 new samples that perform network communication over TCP and UDP, using LwIP over Ethernet as network layer protocol. They were developed based on the demo code for the STM32F429 chip. We integrated modified AFL as the fuzzing engine. Due to the randomness of fuzzing, we conducted 5 trials of 24-hour fuzzing for each test sample to keep the experiment in line with community guidelines [22]. We used the same random value as the initial seed for each test target (P^{2}IM, *μEmu*, *SEmu*). Where applicable, we used the default configurations published alongside the released tools.

Coverage Comparison. The results of the fuzzing experiment is shown in Table 2. We list the median basic block (BB) coverage in the 5 trials, along with the p-value of the experiments. We observe obvious improvement in code coverage over P^{2}IM and *μEmu* in `Soldering_Iron` and four TCP/UDP samples. The reason is that neither P^{2}IM nor *μEmu* supports DMA and thus Ethernet. Although *μEmu* is able to go through the initialization of the Ethernet samples, it failed to cover the main firmware logic which depends on DMA for data transmission. For the firmware that can be tested, *SEmu* yields 5.1% and 6.7% more code coverage on average than P^{2}IM and *μEmu*, respectively. For each sample, we also list the p-value of the tests. They are measured using two-side Mann-Whitney U test for *SEmu* vs. P^{2}IM, and *SEmu* vs. *μEmu*.

The coverage improvement seems insignificant. However, we note that *SEmu* achieves better path fidelity and so does not explore many error handling functions. For example, some peripherals like I2C use two separated interrupts to deal with normal and error signals. The error interrupt only occurs on hardware fault. However, P^{2}IM and *μEmu* still trigger these error interrupts periodically, which should never happen without incurring physical failure. In addition, *SEmu* only fuzzes real I/O interfaces, not the status/control registers which existing work also fuzzes. Because of these reasons, *SEmu* achieves less code coverage on some firmware (*e.g.,* Steering_Control). We argue this is a good indicator of higher emulation fidelity.

Crashes/Hangs. In Table 2, we list the number of crashes and hangs reported by AFL in the column **#Crashes/#Hangs**. To verify the results, for each reported crash/hang, we replayed the trigger testcase to the corresponding emulator, and collected execution traces. If two execution traces are the same, we consider it as duplicate. If we did not observe the crash/hang report during replay, we consider it a false positive. We list the unique crashes/hangs in the column with the header **Unique**, and the number of false crashes/hangs in the column with the header **#False Crashes/Hangs**.

SEmu successfully reproduced all the bugs mentioned in previous work, but did not report any false crashes or hangs. In fact, many crash reports raised in existing work were duplicated or false positives. The reasons are two-fold. First, existing work randomly delivers interrupts, which causes significant changes in the edge coverage of two executions that suffer from the same crash. AFL would thus mistakenly recognize them as two different crashes. Second, as indicated before, due to inaccurate emulation in P^{2}IM and *μEmu*, there are many infeasible paths being explored, leading to considerable occurrences of false positives.

8.6 Compliance Check Test

With semantic information extracted from the specification, and benefited from the high fidelity emulation, *SEmu* makes another kind of security analysis, namely behavior compliance check, possible. In a compliance check, we check whether the implementation of peripheral drivers follows the descriptions from the chip manuals. Specifically, we collect the peripheral access history and match

Table 2: Fuzzing and Compliance Check Results (- indicates the emulator cannot support fuzzing that firmware)

		Median BB Coverage			*p*-value		#Crashes/#Hangs				#False Crashes/Hangs			Compliance Violation
Firmware	**MCU**	***SEmu***	P^{2}IM	*µEmu*	*to* P^{2}IM	*to µEmu*	**Unique**	***SEmu***	P^{2}IM	*µEmu*	***SEmu***	P^{2}IM	*µEmu*	***SEmu***
CNC	STM32F429	**30.90%**	34.31%	28.52%	0.21	0.30	**0/0**	**0/0**	1/3	0/0	**0/0**	1/3	0/0	**None**
Console	K64F	**30.20%**	33.12%	29.31%	<0.01	<0.01	**0/0**	**0/0**	1/3	0/0	**0/0**	1/3	0/0	**None**
Drone	STM32F103	**57.59%**	46.67%	53.48%	<0.01	<0.01	**0/0**	**0/0**	1/7	0/0	**0/0**	1/7	0/0	**None**
Gateway	STM32F103	**36.52%**	36.20%	36.66%	0.92	0.23	**1/0**	**3/0**	136/254	3/0	**0/0**	7/254	0/0	**None**
HeatPress	SAM3X8E	**28.13%**	29.32%	26.88%	<0.01	<0.01	**2/0**	**9/0**	161/48	4/0	**0/0**	8/48	0/0	**None**
PLC	STM32F429	**23.13%**	22.14%	19.84%	<0.01	<0.01	**5/0**	**13/0**	85/27	65/0	**0/0**	5/27	0/0	**None**
Reflow_Oven	STM32F103	**35.67%**	27.97%	29.84%	<0.01	<0.01	**0/0**	**0/0**	1/0	0/0	**0/0**	1/0	0/0	**None**
Robot	STM32F103	**39.99%**	36.90%	35.91%	<0.01	<0.01	**0/0**	**0/0**	1/0	0/0	**0/0**	1/0	0/0	**R2(B)**
Soldering_Iron	STM32F103	**48.92%**	37.60%	35.03%	<0.01	<0.01	**0/0**	**0/0**	1/5	38/7	**0/0**	1/5	38/7	**R2(A)**
Steering_Control	SAM3X8E	**26.65%**	27.07%	27.57%	<0.01	<0.01	**0/0**	**0/0**	1/6	3/0	**0/0**	1/6	2/0	**None**
RF_Door_Lock	STM32L152	**21.20%**	-	20.86%	-	<0.01	**2/0**	**13/0**	-	119/0	0/0	-	0/0	**R2(A)**
Thermostat	STM32L152	**23.59%**	-	22.54%	-	<0.01	**1/0**	**7/0**	-	97/0	0/0	-	0/0	**R2(A)**
LwIP_TCP_Client	STM32F429	**29.45%**	-	-	-	-	**0/0**	**0/0**	-	-	**0/0**	-	-	**None**
LwIP_TCP_Server	STM32F429	**29.40%**	-	-	-	-	**0/0**	**0/0**	-	-	**0/0**	-	-	**None**
LwIP_UDP_Client	STM32F429	**29.75%**	-	-	-	-	**0/0**	**0/0**	-	-	**0/0**	-	-	**None**
LwIP_UDP_Server	STM32F429	**30.36%**	-	-	-	-	**0/0**	**0/0**	-	-	**0/0**	-	-	**None**

it with the expected access constraints learned by NLP. In our prototype, we implemented a dynamic compliance check module that checks two rules.

R1: Peripheral State Verification. The firmware should only access certain registers after checking the status of another register. This simple rule applies to many I/O operations. For example, the firmware first checks whether the hardware is ready by polling the state from a status register. Only if a particular value is returned would it continue to access the data register. R1 is violated if the firmware directly accesses the data register.

R2: Interrupt Activation Consistency. To enable an interrupt, the firmware not only needs to activate the configuration local to the peripheral, but also to enable the corresponding interrupt source by setting the Interrupt Set-Enable Registers (`ISERx`) in the global interrupt manager, namely NVIC. A violation happens when: **R2(A):** the firmware enables the interrupt in NVIC, but not in the local peripheral controller. **R2(B):** the firmware enables the local peripheral controller, but not in NVIC.

8.6.1 Experiment Results. We performed compliance check against the same set of firmware samples used in the previous section, and the unit-test samples of P^{2}IM. The results are as shown in the last column of Table 2.

R1 Violation. We observed several R1 violations in the SAM3X HAL driver code. We later confirmed the root cause to be race conditions of peripheral access. For example, the drive code for UART of SAM3X MCU checks the `TXRDY` field before data transmission. However, it also allows a UART interrupt to happen between `TXRDY` verification and data transmission, during which the interrupt handler conducts an independent data transmission. When the interrupt returns, the previous `TXRDY` verification becomes invalided and the following data transmission would fail. We found similar issues with the SPI unit-test sample on STMF103 MCU.

R2 Violation. Chip vendors often provide a HAL library for developers. It abstracts away hardware details and provides a unified API to access peripheral functions. However, not all HALs support the NVIC register configuration. This is because the IRQ number assignment is chip-specific, and it is developer's responsible to configure NVIC. This assumption gives rise to the violation of R2. In particular, the developer may forget configuring NVIC. For example, we found the `Robot` firmware invokes the HAL function `HAL_TIM_Base_Start_IT()` to set the `TIM2_DIER` register to enable the `Timer2` interrupt. However, it does not enable the corresponding interrupt number (28) in the NVIC registers. As a result, the interrupt can never be delivered. This happens to the original K64F Timer unit test sample too. After manual verification, we found the test code enables a non-existing interrupt channel for the K64F Timer, which has been confirmed and corrected by P^{2}IM authors [2]. We observed similar violations happened to `Soldering_Iron`, `RF_Door_Lock`, and `Thermostat`.

9 LIMITATIONS AND DISCUSSION

NLP Limitations. The state-of-the-art NLP tools have limitations in handling references across different sentences and complex sentences with underlying or nested conditions. These limitations cause faulty rules as discussed in Section 8.2. However, unlike other applications, *SEmu* suffers less from these limitation because 1) actions are often explicitly expressed as executing several simple register assignments and no cross-section reference is used; 2) named entities are formal expressions in chip manuals, alleviating co-reference problems (*e.g.,* we use approximate string matching to unify named entities); 3) complex conditions often depend on certain hardware signals with implicit semantics, which can often be modelled by a default behavior (*e.g.,* the always "set" heuristic we adopted in the F103 ADC example in Section 6). We also plan to leverage new advances in NLP techniques to improve *SEmu* in the future. For example, *SEmu* can greatly benefit from the improvement on co-reference resolution in deciding the conditions of hardware triggered signals.

Manual Efforts. There are three major sources of manual efforts. First, for each firmware, the developers need to identify the underlying MUC chip and configure the memory mapping information (*e.g.,* the ranges of memory and MMIO) for the emulator. Second, the developers need to manually copy some required sections in the chip manuals and input them to the NLP engine. Currently, register memory map, field description, interrupt vector assignment table and DMA channel assignment table, or equivalent sections are needed. Third, diagnosing faulty C-A rules requires two-fold manual efforts. 1) Test firmware and test-cases need to be prepared; 2) When an invalid state is detected, the developers need to read and comprehend the specification to fine-tune the faulty C-A rules. The latter cannot be avoided since there are many domain terminologies which cannot be understandable by NLP. For example, the F1XX

manual mentions that I2C can enter different modes depending on the LSB of the address byte. The knowledge that LSB refers to the "least significant bits" must be provided by experts.

To evaluate the manual effort needed in C-A rule diagnosis, we invited three embedded system developers with basic domain knowledge on MCU. After reading through the relevant chapters of the manual, two hours were needed to fix the problematic C-A rules for F103 I2C on average. In comparison, we estimate that more than a week is needed for an expert to write an I2C backend for QEMU.

10 RELATED WORK

10.1 Firmware Emulation

Relying on real hardware for dynamic analysis incurs many problems [20, 21, 23, 28], such as poor performance and low scalability. Emulation is an effective way to address these issues. The key challenge of emulating firmware is how to properly model the peripheral behaviors so that the emulated execution can be similar to that on real hardware.

High-level emulation solutions [5, 7, 8, 24, 36] avoid emulating code related to peripheral by hooking into high-level libraries (*e.g.,* hardware abstraction layer) and implementing equivalent logic on the native machine. High-level emulation achieves good fidelity as *SEmu* does. However, these approaches completely skip the peripheral logic in firmware, and therefore cannot find problems with peripheral drivers. Furthermore, developers do not always use a high-level abstraction library for performance consideration.

Firmware-guided solutions [3, 5, 12, 19, 39, 49] run the whole target firmware in the emulator. Based on the strategy, they can be further classified into three types: access-pattern-based, symbolic-execution-based, and learning-based. By observing the peripheral access pattern, P^{2}IM infers register types (*i.e.,* `CR`, `SR`, `C&SR` and `DR`), then it uses heuristics to generates responses based on the register type information. However, as mentioned before, it suffers from the register mis-categorization problem. Besides, when heuristic is unavailable, P^{2}IM blindly searches for appropriate responses with limited search spaces. Symbolic-execution-based solutions [3, 19, 49] address the aforementioned problems by reasoning about how responses from peripheral can influence firmware execution. A key limitation of these solutions is that they rely on heuristics to decide the path to take. Indeed, firmware does not contain enough information to guide the emulation. PRETENDER [16] and Conware [39] create peripheral models by learning from the real interactions between the hardware and firmware. As such, they require a hardware dependent recording phase, reducing the scalability. Compared with firmware-guided solutions, our approach is guided by peripheral specifications. It achieves higher emulation fidelity without requiring real hardware.

10.2 NLP-based Rule Extraction

Extracting rules from specifications to solve security problems is not new. SmartAuth [46] learns the policy correlations (entity, context and action, condition) from IoT app descriptions, and verifies whether the code implementation of IoT apps follows the policy. ARE [13] automatically discovers IoT devices by generating (device detection) rules using application-layer data and product descriptions. iRuler [47] uses NLP techniques to infer trigger-action information flows and discover inter-rule vulnerabilities within IoT deployments. NLP techniques are also used to automatically collect and analyze IoT security reports to extract vulnerability-specific features [14]. Bookworm Game [6] uses NLP techniques to identify risky operations from LTE (Long-Term Evolution) documentation in telecommunication. Then it extracts information to construct testcases that can satisfy the conditions to trigger risky operations. Access control policy [48] and access control rules [38] can also be extracted from use case descriptions.

Our work automatically constructs peripheral models for security-analysis-driven firmware emulation. This is a different NLP application from existing work. This specific problem introduces a few unique challenges. First, a C-A rule can be triggered in different ways. We must categorize the extracted rules so that they can be checked only when needed. Second, to support firmware emulation, the semantics of the extracted rules must be formalized into concrete values and Boolean operators so that the peripheral states can be programmatically maintained. Third, in order to handle hardware-generated signals and interrupt-related actions, we must incorporate MCU-specific domain-knowledge into NLP. Most importantly, existing work does not need to comprehensively extract relevant rules, but *SEmu* must ensure that the extracted rules are adequately complete. For example, in Bookworm Game [6], the more context information it can extract, the more risky operations it could identify. Even if the extracted information items are only a subset, their approach can still outperform solutions not guided by NLP techniques. In contrast, if *SEmu* does not achieve adequate completeness of the relevant C-A rules (including the chained rules), firmware emulation will very likely fail. This key difference motivates the unique rule diagnosis mechanism of *SEmu*.

11 CONCLUSION

Instead of proposing yet another firmware-guided emulation solution, in this work, we propose the first specification-based firmware emulation solution. The new approach leverages NLP techniques to translate peripheral behaviors (specified) in human language (documented in chip manuals) into a set of structured condition-action rules. By properly executing and chaining these rules at runtime, we can dynamically synthesize a peripheral model for each peripheral accessed during firmware execution. With the help of machine-aided rule diagnosis, our evaluation confirmed that our prototype achieves 100% emulation fidelity compared with real devices under the proposed fidelity measurement. In comparison, the fidelity achieved by existing work varies between 73% and 86%. With better fidelity, our solution improves fuzzing efficiency: no false crashes and hangs were observed during fuzz testing. With much higher emulation accuracy, we also designed a new dynamic analysis task to perform driver code compliance checks against the specification. We found some non-compliance which we later confirmed to be bugs caused by race condition.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This project was supported in part by NSF (CNS-1814679, CNS-2019340), Cisco Research, NSFC (U1836210) and the Key Research and Development Science and Technology of Hainan Province (ZDYF202012, GHYF2022010).

REFERENCES

[1] Feng Bo, Muench Marius, Zhou Wei, and Scharnowski Tobias. 2022. P2IM Unit test Errata. https://github.com/RiS3-Lab/p2im-unit_tests#errata. (2022). Last accessed: 2022-08-01.

[2] bofeng17 and lfz5092. 2022. P2IM Errata. https://github.com/RiS3-Lab/p2im-unit_tests/commit/cdfd9bbc72e1cc87a1f5d3905804ae1f91539beb. (2022). Last accessed: 2022-08-01.

[3] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference*. 746–759.

[4] Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 740–750.

[5] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware.. In *NDSS*, Vol. 16. 1–16.

[6] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. 2021. Bookworm Game: Automatic Discovery of LTE Vulnerabilities Through Documentation Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1197–1214. https://doi.org/10.1109/SP40001.2021.00104

[7] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium*. 1–18.

[8] Andrei Costin, Apostolis Zarras, and Aur'elien Francillon. 2016. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 437–448.

[9] Cyberhaven. 2022. Analyzing large programs using concolic execution. http://s2e.systems/docs/Howtos/Concolic.html. (2022).

[10] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. 2021. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 687–701.

[11] Bo Feng. 2021. P2IM unit test samples. https://github.com/RiS3-Lab/p2im-unit_tests/tree/30e6aec9f5c44f11b8072bf597eb80729dad417d. (2021).

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

[13] Xuan Feng, Qiang Li, Haining Wang, and Limin Sun. 2018. Acquisitional rule-based engine for discovering internet-of-things devices. In *27th USENIX Security Symposium (USENIX Security 18)*. 327–341.

[14] Xuan Feng, Xiaojing Liao, X Wang, Haining Wang, Qiang Li, Kai Yang, Hongsong Zhu, and Limin Sun. 2019. Understanding and securing device vulnerabilities through automated bug report analysis. In *SEC'19: Proceedings of the 28th USENIX Conference on Security Symposium*.

[15] FRDM-K64F. 2021. Freedom Development Platform. https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F. (2021). Last accessed: 2022-05-01.

[16] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 135–150.

[17] Eric Gustafson and Moritz Schloegel. 2021. Pretender firmware samples. https://github.com/ucsb-seclab/pretender/tree/master/test_programs/max32600. (2021).

[18] Hubert Högl and Dominic Rath. 2006. Open on-chip debugger–openocd–. *Fakultat fur Informatik, Tech. Rep* (2006).

[19] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*.

[20] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*.

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

[22] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

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

[24] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. 2021. From Library Portability to Para-rehosting: Natively Executing Open-source Microcontroller OSs on Commodity Hardware. In *28th Network and Distributed System Security Symposium (NDSS '21)*. The Internet Society.

[25] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[26] MarketWatch. 2022. Microcontroller Unit (MCU) Market Size 2022. https://www.marketwatch.com/press-release/microcontroller-unit-mcu-market-size-2022-industry-growth-latest-update-with-technological-advancement-emerging-trends-business-opportunity-sales-revenue-gross-margin-and-forecast-to-2030-2022-06-06. (2022). Last accessed: 2022-08-01.

[27] Microchip. 2021. SAM3X/SAM3A Series Atmel SMART ARM-based MCU DATASHEET. https://ww1.microchip.com/downloads/en/devicedoc/atmel-11057-32-bit-cortex-m3-microcontroller-sam3x-sam3a_datasheet.pdf. (2021). Last accessed: 2022-07-01.

[28] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, Vol. 18. 1–11.

[29] Ethan Nam. 2021. Understanding the Levenshtein Distance Equation for Beginners. https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0. (2021). Last accessed: 2022-05-01.

[30] Nordic. 2021. nRF52840 Product Specification v1.7. https://infocenter.nordicsemi.com/pdf/nRF52840_PS_v1.7.pdf. (2021). Last accessed: 2022-05-01.

[31] NXP. 2014. FRDM-K64F Freedom Module User's Guide. https://www.mouser.com/datasheet/2/302/NXP_01112019_FRDM_K64F-1891948.pdf. (2014). Last accessed: 2022-05-01.

[32] NXP. 2014. K64 Sub-Family Reference Manual. https://www.mouser.com/datasheet/2/813/K64P144M120SF5RM-1074828.pdf. (2014). Last accessed: 2022-05-01.

[33] QEMU-6.1. 2022. QEMU backend for STMF405/205 ADC. https://github.com/qemu/qemu/blob/stable-6.1/hw/adc/stm32f2xx_adc.c#L75. (2022). Last accessed: 2022-08-01.

[34] QEMU-6.1. 2022. QEMU backend for STMF405/205 USART. https://github.com/qemu/qemu/blob/stable-6.1/hw/char/stm32f2xx_usart.c#L67. (2022). Last accessed: 2022-08-01.

[35] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

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

[37] Abragam Siyon Sing. 2021. State Diagram and state table with solved problem on state reduction. https://www.electrically4u.com/state-diagram-and-state-table/. (2021).

[38] John Slankas, Xusheng Xiao, Laurie Williams, and Tao Xie. 2014. Relation extraction for inferring access control rules from natural language artifacts. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 366–375.

[39] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. 2021. Conware: Automated Modeling of Hardware Peripherals. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 95–109.

[40] STMicroelectronics. 2020. RM0038 Reference manual. https://www.st.com/resource/en/reference_manual/cd00240193-stm32l100xx-stm32l151xx-stm32l152xx-and-stm32l162xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf. (2020). Last accessed: 2022-05-01.

[41] STMicroelectronics. 2021. RM0008 Reference manual. https://www.st.com/resource/en/reference_manual/cd00171190-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf. (2021). Last accessed: 2022-05-01.

[42] STMicroelectronics. 2021. RM0090 Reference manual. https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf. (2021). Last accessed: 2022-05-01.

[43] STMicroelectronics. 2021. STM32 Nucleo-32 boards User Manul. https://www.st.com/resource/en/user_manual/dm00231744-stm32-nucleo32-boards-mb1180-stmicroelectronics.pdf. (2021). Last accessed: 2022-05-01.

[44] STMicroelectronics. 2021. STM32 Nucleo-64 development board with STM32F103RB MCU. https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-f103rb.html. (2021). Last accessed: 2022-05-01.

[45] STMicroelectronics. 2022. STM32Cube MCU Package for STM32F4 series. https://www.st.com/en/embedded-software/stm32cubef4.html#get-software. (2022). Last accessed: 2022-07-01.

[46] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. Smartauth: User-centered authorization for the internet of things. In *26th USENIX Security Symposium (USENIX Security 17)*. 361–378.

[47] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A Gunter. 2019. Charting the attack surface of trigger-action IoT platforms. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1439–1453.

[48] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. 2012. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[49] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *30th USENIX Security Symposium (USENIX Security 21)*.

[50] Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 434–443.