PHI: Pseudo-HAL Identification for Scalable Firmware Fuzzing

Seyeon Jeong[1,2]⋆, Eunbi Hwang[1,⋆], Yeongpil Cho[3], and Taekyoung Kwon[1]⋆⋆

[1] Graduate School of Information, Yonsei University, Seoul 03722, South Korea
[2] Suresofttech Inc., Seongnam-si, Gyeonggi 13453, South Korea
best6653@gmail.com, {ebhwang95, taekyoung}@yonsei.ac.kr
[3] Hanyang University, Seoul 08826, South Korea
ypcho@hanyang.ac.kr

Abstract. Firmware fuzzing aims to detect vulnerabilities in firmware by emulating peripherals at different levels: hardware, register, and function. HAL-Fuzz, which emulates peripherals through HAL function handling, is a remarkable firmware fuzzer. However, its effectiveness is confined to firmware solely relying on HAL functions, and it necessitates intricate firmware information for best outcomes, thereby limiting its target firmware range. Notably, in commercial firmware, both HAL and non-HAL (which we call "pseudo-HAL") functions are prevalent. Identifying and addressing both is crucial for comprehensive peripheral control in fuzzing. In this paper, we present PHI, a tool designed to identify HAL and pseudo-HAL functions at the register level. Using PHI, we develop PHI-Fuzz, an enhanced firmware fuzzer operating at the function level. This fuzzer efficiently manages HAL and pseudo-HAL functions, demanding minimal prior knowledge yet delivering substantial results. Our evaluation demonstrates that PHI identifies HAL functions accessing the MMIO range as effectively as LibMatch of HAL-Fuzz, while overcoming its constraints in detecting pseudo-HAL functions. Significantly, when benchmarked against HAL-Fuzz, PHI-Fuzz showcases superior bug-finding capabilities, uncovering crashes that HAL-Fuzz missed.

Keywords: Security, Firmware, Fuzzing, Hardware Abstraction Layer

1 Introduction

Embedded devices play a crucial role in various applications, including the Internet of Things (IoT), aviation, and weapons systems. According to *State of IoT—Spring 2023* [1] report, there was an 18% growth in the number of global IoT connections during 2022, resulting in a total of 14.3 billion active IoT endpoints. However, when compared to the total vulnerabilities discovered, firmware vulnerabilities have consistently accounted for about 2% each year since 2017, and as of 2023, 2.41% of firmware vulnerabilities have been identified [2].

⋆ Equal contribution.
⋆⋆ Corresponding author.

Firmware vulnerabilities, which can result from system crashes, reboots, and hangs, are exploitable by attackers aiming to compromise embedded devices. This poses a significant risk to society, thus necessitating dynamic analysis and proactive detection through firmware fuzzing [14, 19, 20].

Fuzzing, a dynamic bug-finding technique, provides random input values to a program and monitors its executions. AFL (American Fuzz Lop) [24] is a coverage-guided fuzzer that has demonstrated high performance in general software fuzzing and can also be utilized for firmware fuzzing on microcontroller units (MCU) [16, 20, 25]. However, exploring firmware vulnerabilities through fuzzing techniques can be challenging, particularly for embedded devices with inherent limitations. To address these challenges, recent firmware fuzzing research has proposed emulation-based fuzzing [10–13, 18, 26]. Firmware emulation enables fuzzing on devices with sufficient power and capacity. Nonetheless, using a general emulator like QEMU [9] can lead to execution failures due to undefined peripheral access during firmware fuzzing. Consequently, how emulators handle peripherals is crucial for successful firmware emulation and fuzzing. Emulation through Hardware-In-The-Loop (HITL) method can result in performance degradation due to communication between hardware and the emulator [19]. Recent studies have focused on peripheral modeling as a way to overcome this limitation. Peripheral modeling techniques can be classified into three types: hardware-level, function-level, and register-level modeling. Function-level and register-level modeling do not require hardware during the modeling phase, resulting in better performance for firmware emulation and fuzzing.

Function-level peripheral modeling involves emulating firmware by hooking a function during emulation and connecting pre-made handlers. Register-level peripheral modeling handles each register during emulation. Compared to register-level modeling, function-level modeling boasts faster processing, as peripheral functions accessing Memory-mapped I/O (MMIO) are processed with a handler. HALucinator, a firmware emulator, implements function-level peripheral modeling using Python handlers achieved through Hardware Abstraction Layer (HAL) function hooking [11]. Building upon this concept, HAL-Fuzz, a firmware fuzzer, integrates HALucinator with UnicornAFL [3]. HALucinator and HAL-Fuzz identify functions to be hooked using LibMatch [4], a HAL function identification tool. Although LibMatch can identify HAL functions, it requires a software development kit (SDK) containing HAL function object files compiled in the same environment as the target firmware. As a result, LibMatch needs extensive information about the firmware despite its limited capabilities in identifying functions.

Many modern firmware implementations utilize not only HAL but also pseudo-HAL functions. Consequently, LibMatch may not fully identify all functions in the firmware, limiting the effectiveness of HALucinator and HAL-Fuzz. Additionally, obtaining detailed information about firmware compilation options can be challenging, and the scripts used in LibMatch are often not openly available. This makes it difficult to use LibMatch in an ideal operating environment. To overcome these limitations, we propose the Pseudo-HAL Identification

(PHI) program, which leverages symbolic execution to identify HAL and pseudo-HAL functions at the register level without relying on specific firmware compilation environments or firmware stripping. Furthermore, we introduce PHI-Fuzz, a function-level firmware fuzzer based on HAL-Fuzz that utilizes PHI's results. With the scalability provided by PHI, PHI-Fuzz can perform more efficient and effective fuzzing compared to existing function-level firmware fuzzers.
Contribution. This paper makes the following contributions.

- **Pseudo-HAL Identification** We propose PHI, a register-level function identification method for more scalable function-level peripheral modeling.
- **PHI-Fuzz** We propose PHI-Fuzz, an enhanced and scalable firmware fuzzer operating at the function level by leveraging PHI.
- For further research, we will release our tool at publication time.

Organization. This paper is organized as follows. Section II provides the necessary background and discusses the existing problems. Section III presents the design of the proposed system. Section IV describes the implementation of the system. Section V presents the evaluation of the system. Section VI provides a discussion of the results and limitations. Section VII reviews the related work. Finally, Section VIII concludes the paper.

2 Motivation

In this section, we briefly discuss the background of firmware fuzzing, identify the challenges of existing techniques, and demonstrate their limitations through a series of experiments.

2.1 Background

Firmware in Embedded Devices Firmware is a type of software that offers low-level control over hardware components, including on-chip and off-chip peripherals, as well as MCUs integrated into embedded systems. Muench et al. [19] classified embedded devices into three categories based on firmware: general OS-based firmware, embedded OS-based firmware, or monolithic firmware. Monolithic firmware (also known as bare-metal firmware) is present in approximately 81% of embedded devices as of 2019 [5]. This firmware type operates by executing simple functions in a continuous loop and is commonly used in small-scale embedded systems. Our study focuses on developing a firmware fuzzing technique that specifically targets monolithic firmware.

Firmware Fuzzing Traditional fuzzing techniques for general software often require instrumentation to observe and analyze the behavior of the tested program. However, firmware fuzzing presents additional challenges due to the high dependency on heterogeneous peripherals and the lack of reliable emulation techniques. Fully emulating firmware, including both the processor and peripherals,

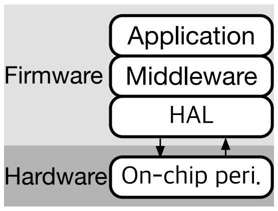


Fig. 1. STM32 firmware architecture

can be a complex and time-consuming process owing to the wide variety of peripherals available. For firmware testing, partial emulation using the hardware-in-the-loop (HITL) method may be slower than the peripheral modeling method, as it may cause a bottleneck in the communication process between the emulator and the actual hardware being emulated [19, 22]. Recently, emulation techniques utilizing peripheral modeling have gained popularity for effective firmware fuzzing [10–13, 18, 21, 26].

HAL(Hardware Abstraction Layer) HAL is a library provided by manufacturers to enhance the convenience of firmware development. By abstracting common functionality for specific devices, HAL makes developers program without relying on a specific hardware target [6]. Since many manufacturers produce various types of hardware, developing firmware based on specific hardware requires a significant loss of productivity to develop firmware that directly accesses the hardware. Using HAL has the advantage of facilitating the development of essential functions when creating firmware. It is presented as higher-layer functions rather than register units, enabling convenient usage through function calls without the need for direct register access. For instance, in implementing the functionality to send data over UART, developers can simply call the *HAL_UART_Transmit()* function without directly manipulating the Data Register. HALucinator [11] leveraged the characteristics of this HAL in firmware emulation. Identified HAL function calls and handled them with pre-made handlers, HALucinator improved emulation efficiency. Unlike HALucinator, which identified HAL functions at the function level, the PHI proposed in this paper detects not only HAL functions but also various library functions for peripherals HAL functions at the register level.

2.2 Problem Definition

A central question this study aimed to address is whether function-level fuzzing, as a peripheral modeling method, is more efficient than register-level fuzzing. We also examined the scalability of current function-level emulation techniques. To answer these questions, we conducted several experiments as part of our research.

Example 1 Firmware execution code

```
1: int main(){
2:     char a[5];
3:     char b = HAL_uart_getc();
4:     a[b] = 1;
5: }
```

Example 2 Firmware execution code

```
1: int main(){
2:     char a[5];
3:     data = HAL_UART_Receive_IT(huart, pData, Size);
4:     strcpy(a, data);
5: }
```

Efficiency of function-level emulation for fuzzing This paper investigates the use of different levels of peripheral modeling for firmware fuzzing, including hardware, function, and register levels. While hardware-level modeling necessitates physical devices, function-level and register-level modeling can be achieved through emulation. To compare the performance of firmware fuzzing at the function and register levels, we conducted an experiment using recent fuzzers, including HAL-Fuzz, P^{2}IM, Fuzzware, and HEFF. HAL-Fuzz employs function-level modeling, while P^{2}IM and Fuzzware utilize register modeling. HEFF uses dual-level modeling at both functional and register levels [15]. We tested these fuzzers on the Drone firmware [12], and the results are presented in Table 9. The experiment indicates that the fuzzing speed of register-level fuzzers (including dual-level fuzzers) is approximately half as fast as the fuzzing speed of HAL-Fuzz, a function-level fuzzer. These results suggest that function-level fuzzing is a more efficient approach.

The difference in fuzzing speed between function-level and register-level fuzzing (including dual-level) is due to the additional processing overhead incurred by register-level fuzzing as it handles all accessed registers (also partially handles accessed registers). Firmware vulnerabilities can arise from processing inputs received through peripherals. We provide two examples of vulnerabilities resulting from buffer overflow in this paper. In Example 1, a vulnerability occurs in line 4, where an external input is received through the HAL function and stored as a variable. In Example 2, an external input is saved as a variable, leading to a vulnerability. While both examples use HAL functions, the vulnerabilities arise outside of the HAL function, not within it. In the above-mentioned case, register-level emulation handles all accesses made inside the HAL function, whereas function-level emulation handles functions with pre-made handlers, thus avoiding any processing overhead.

The necessity of identifying pseudo-HAL functions. Figure 1 illustrates the structure of STM32 firmware, where the HAL acts as an intermediate layer

Table 1. Peripheral related functions in CNC firmware

Firmware	Pseudo-HAL	HAL
CNC	`dirn_wr`	`HAL_DeInit`
	`enable_tim_clock`	`HAL_DisableCompensationCell`
	`enable_tim_interrupt`	`HAL_EnableCompensationCell`
	`enable_usart_clock`	`HAL_GPIO_DeInit`
	`g540_timer_init`	`HAL_GPIO_EXTI_IRQHandler`
	`g540_timer_start`	`HAL_GPIO_Init`
	`g540_timer_stop`	`HAL_GPIO_ReadPin`
	`gpio_clr`	`HAL_GPIO_TogglePin`
	`gpio_init`	`HAL_GPIO_WritePin`
	`gpio_rd`	`HAL_Init`
	`gpio_set`	`HAL_RCC_ClockConfig`
	`gpio_toggle`	`HAL_RCC_DeInit`
	`mc_dwell`	`HAL_RCC_GetHCLKFreq`
	`set_step_period`	`HAL_RCC_GetOscConfig`
	`set_step_pulse_delay`	`HAL_RCC_GetPCLK1Freq`
	`set_step_pulse_time`	`HAL_RCC_GetPCLK2Freq`
	`step_isr_disable`	`HAL_RCC_GetSysClockFreq`
	`step_isr_enable`	`HAL_RCC_MCOConfig`
	`step_timer_init`	`HAL_RCC_NMI_IRQHandler`
	`step_wr`	`HAL_RCC_OscConfig`
	`SystemClock_Config`	
	`SystemCoreClockUpdate`	
	`SystemInit`	
	`TIM2_IRQHandler`	
	`usart_getc`	
	`usart_init`	
	`usart_putc`	
	`usart_tstc`	
Total(#)	28	20

between hardware and software, directly writing values to MCU registers or controlling peripheral devices. The HAL is a universal library commonly employed by developers to manage peripheral devices in firmware implementation. Tools like HALucinator and HAL-Fuzz are used to identify and hook these HAL functions for handling. The HAL function identification program proposed in [11], called LibMatch, is currently employed for this purpose. This enables firmware to operate without requiring physical peripheral devices or separate peripheral emulations. However, LibMatch has two significant limitations due to its reliance on a context-matching technique between the target firmware and the HAL function object file to extract HAL function information.

A lot of information is required. The first limitation of LibMatch is that it necessitates the SDK (object file of the HAL functions) to be compiled in an

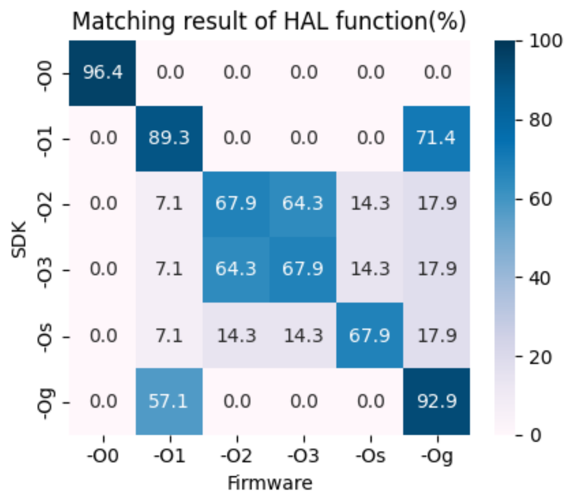


Fig. 2. Result of libmatch HAL function identification according to SDK and firmware combination by compile optimization level

environment with the same compiler version and optimization level as the target firmware. FIGURE 2 displays the LIBMATCH function identification results for six types of compile optimization levels of the target firmware and their corresponding SDKs. The x-axis represents the optimization options for firmware, while the y-axis represents the optimization options for the SDK. For example, in Figure 2, the matrix (0,0) represents 96.4% of the matching HAL function ratio when the firmware is built with the -O0 option and the SDK is built with the -O0 option, using the libmatch extraction method. When the optimization levels match (6 out of 36), a high matching rate ranging from 67.9% to 96.4% is achieved. However, in most cases where the optimization levels do not match (30 out of 36), function search is either impossible or, even if a match is identified, the matching rate is below 20%. This indicates that Libmatch has a high dependency on the SDK files. If it fails to find an SDK that matches the optimization options of the target firmware, the matching ratio of HAL functions decreases.

Unidentified functions exist. The second limitation of LIBMATCH is that it can only identify HAL functions, as the required SDK file contains only HAL function information. Consequently, functions other than HAL functions cannot be identified by LIBMATCH. However, as demonstrated in the CNC [7] firmware example in Table 1, not all firmware exclusively depends on HAL functions to control their peripherals. In such cases, developers define and utilize functions that behave like HAL but can be controlled in smaller units for convenience. These functions, referred to as pseudo-HAL functions in this study, perform functions using registers assigned to peripheral devices while accessing within the range of the HAL functions and MMIO. Therefore, for scalable function-level firmware fuzzing, it is crucial to identify both HAL and pseudo-HAL functions.

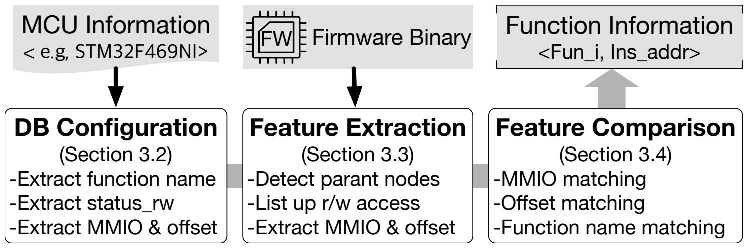


Fig. 3. PHI system flow

2.3 Our approach

We propose the use of pseudo-HAL function identification for effective and scalable firmware fuzzing at the function level. Pseudo-HAL functions are identified based on register access patterns at the register level. This can be accomplished through symbolic execution of MMIO and identifying characteristic offset information for each function. This approach reduces the reliance on the SDK compilation environment and enables fuzzing of a wider range of firmware than HAL-Fuzz. In the next section, we will provide a detailed description of our PHI system.

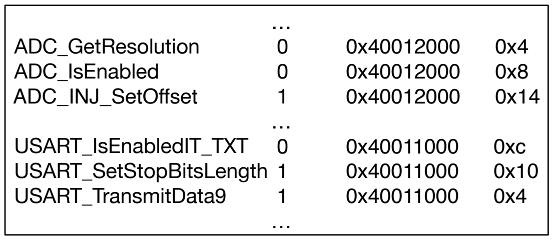
3 System Design

3.1 System Overview

In this section, we present an overview of the PHI (Pseudo-HAL Identification) system, which involves a two-input, three-step process, as illustrated in Figure 3. The user provides the target firmware and the corresponding MCU (Microcontroller Unit) name as inputs. The MCU name is used for selecting the appropriate DB (Database) file, while the firmware is utilized for feature extraction to identify functions related to peripheral devices. The PHI process comprises three steps: DB configuration, feature extraction, and feature comparison. DB configuration (Section 3.2) is the first step, which involves creating a DB for each MCU prior to the PHI operation and selecting the appropriate DB based on the input MCU name. The second step, feature extraction (Section 3.3), extracts the function features from the firmware using symbolic execution. This step is the most computationally intensive and involves the extraction of three features for each peripheral access. In the final step, feature comparison (Section 3.4), the functions used in the firmware are identified by matching the extracted features with the DB. The extracted files in this step are utilized for fuzzing.

3.2 DB Configuration

The process of configuring the DB includes two primary steps: DB creation and DB selection. DB creation involves extracting the features of peripheral



	...		
ADC_GetResolution	0	0x40012000	0x4
ADC_IsEnabled	0	0x40012000	0x8
ADC_INJ_SetOffset	1	0x40012000	0x14
	...		
USART_IsEnabledIT_TXT	0	0x40011000	0xc
USART_SetStopBitsLength	1	0x40011000	0x10
USART_TransmitData9	1	0x40011000	0x4
	...		

Fig. 4. Example of DB

functions used in each MCU from MMIO (Memory-Mapped Input/Output) and offset that can be called from the embedded board, and converting them into a database. This process is essential for obtaining the necessary information to accurately map the functions used in the firmware to the MCU. It involves analyzing the registers used and their corresponding states, as well as dividing the base address and offset of each peripheral device to enable further classification. As a result, the database structure can be represented as <*func_i, state_rw, peri_addr, offset*>. FIGURE 4 illustrates an example of a database (DB). DB includes the name of low-level functions (*func_i*), whether the function involves reading or writing to the MMIO registers (*state_rw*), the MMIO address associated with the function (*peri_addr*), and the register access offset (*offset*). For the indication of reading or writing to MMIO registers, 0 represents the state of reading from the MMIO register, and 1 represents the state of writing to the MMIO register. In the first row of the FIGURE 4, *ADC_GetResolution* represents the function name, 0 indicates reading from the MMIO register, `0x40012000` indicates the base address of peripheral and `0x4` indicates the offset for accessing the MMIO register.

DB selection, on the other hand, is the process of selecting the appropriate DB based on the MCU name input for PHI. This step is crucial for effective and accurate PHI operation. These DBs are stored in a single folder, and DB selection is the process of selecting a DB corresponding to the entered MCU name. The reason for configuring various DBs is that the register addresses used for each MCU are different, and selecting the correct DB ensures the proper mapping of peripheral functions to the specific MCU.

3.3 Feature Extraction

The feature extraction step extracts the features of functions called when the target firmware is executed using symbolic execution, a static analysis technique. Typically, to identify functions at the function-level, an object file containing function information is necessary, as in the case of LIBMATCH results. However, this paper proposes a register-level function detection approach that extracts function features from all register-level accesses without requiring detailed function information, such as function names. As a result, we leverage symbolic

Table 2. Information of USART

Register		Offset
Status Register	`SR`	`0x00`
Data Register	`DR`	`0x04`
Baud Rate Register	`BRR`	`0x08`
	`CR1`	`0x0C`
Control Register	`CR2`	`0x10`
	`CR3`	`0x14`
Guard Time and Prescalar Register	`GTPR`	`0x18`

execution to identify functions at the register level without relying on detailed information, instead of using a matching method that requires such information. This approach is possible because peripheral registers in firmware are assigned to specific memory ranges, such as the MMIO range of `0x40000000`–`0x5fffffff` for ARM Cortex-M4 MCUs, for example.

Consider the case of USART, which manages asynchronous serial communication between computers. In an ARM Cortex-M4 MCU, the peripheral base address for USART is `0x40011000`, and offsets such as SR, DR, BRR, CR, and GTPR are allocated to it, as shown in Table 2. By utilizing these offsets and their corresponding USART functions, which control USART using the related registers, it is possible to identify functions at the register level without the need for detailed information, such as function names. MMIO ranges, peripheral base addresses, and offset information can be obtained from the datasheet for each MCU, facilitating the construction of this information. Therefore, to extract the features of functions related to firmware peripherals, the following steps are performed:

1. List the functions that access the MMIO range.
2. Check the base address and offset used by each function.
3. Record whether the function reads or writes to that memory.

To accomplish this, the top-level parent node is first extracted from the target firmware. Then, the function call flow within the firmware is checked, starting from all parent nodes. All accesses that read or write memory information within the MMIO address range are recorded. These accesses are listed by creating the tuple <*instruction address (ins_addr), block address (block_addr), state_rw, peri_addr, and offset*>. Typically, functions can access the MMIO range multiple times, and memory reads/writes can occur sequentially. If a function has a continuous sequence of the same type of operation, such as read/read/read/... or write/write/write/..., the sequence of accesses is summarized into a single input. However, if both read and write operations occur in the same function with the same offset, they are summarized as a write operation because the same offset is read and written when writing to a specific register for a function.

3.4 Feature Comparison

In the feature comparison step, a list of functions for fuzzing is extracted by matching the feature extraction results, which consist of instruction address, block address, status (read or write), peripheral base address, and offset, with the previously constructed database. These function names are used as keys when connecting to a function handler after function hooking. The corresponding function search result field is the same as that of <*func_i, ins_addr*>. In this step, the corresponding results are extracted to a file and used for function hooking during fuzzing.

4 Implementation

In this study, we implemented PHI, PHI-Fuzz, and a handler. PHI takes the firmware binary and the name of the MCU on which the firmware is loaded as input, then selects the DB corresponding to the MCU name. The PHI is implemented as a Python script consisting of 479 lines, which configures the function information DB, totaling 972 lines of code.

To configure the DB and identify the pseudo-HAL, PHI utilizes angr [8], a symbolic execution tool. The angr functions used include Control-Flow Graph (CFG) analysis and Data Dependency Graph (DDG) results. The CFG functions were divided into CFGFast and CFGEmulated. CFGFast was employed to extract the parent node, while CFGEmulated (with a call depth of 7) was used to extract the DDG.

PHI-Fuzz is implemented based on HAL-Fuzz and receives the PHI result as an addr.yaml file, saves it, and fuzzes the target firmware through a modified handler. The essential handler functions for fuzzing were implemented by adding them to the existing HAL function handler file. Specifically, the existing HAL function handler was connected with the pseudo-HAL function, which played a similar role, to enable fuzzing. Functions discovered through PHI that could not be replaced with existing functions were implemented and added to the existing handler file.

5 Evaluation

The evaluation of PHI-Fuzz was experimentally conducted to answer the following research questions:

- **RQ1:** How scalable is a PHI that uses only firmware images for identification?
- **RQ2:** How effective is the PHI in terms of function identification?
- **RQ3:** How good is the PHI-Fuzz in Bug finding?

Table 3. Firmware tested in Section 5.2, 5.3, 5.4

Firmware	**MCU**	**OS**	**Library**	**Peripherals**			
				GPIO	UART	I2C	SPI
UART_transmit	STM32F469NI	Baremetal	HAL	✓	✓		
UART_receive		Baremetal		✓	✓		
I2C_receive		Baremetal		✓	✓	✓	
SPI_receive		Baremetal		✓	✓		✓
UART _HyperTerminal_IT [11]		Baremetal		✓	✓		
Drone [12]	STM32F103RB	Baremetal	HAL	✓	✓	✓	
CNC [12]		Baremetal	HAL, Pseudo-HAL	✓	✓	✓	
Baremetal_I2C	STM32F469NI	Baremetal	Pseudo-HAL	✓	✓	✓	
FreeRTOS_I2C		FreeRTOS		✓	✓	✓	
Baremetal_UART		Baremetal		✓	✓		
FreeRTOS_UART		FreeRTOS		✓	✓		
RIOT_I2C_receive		RIOT OS		✓		✓	
RIOT_I2C_transmit		RIOT OS		✓	✓	✓	
RIOT_SPI_receive		RIOT OS		✓			✓
RIOT_UART	STM32F103RB	RIOT OS	Pseudo-HAL	✓	✓		
RIOT_SPI		RIOT OS		✓	✓		✓
RIOT_I2C		RIOT OS		✓	✓	✓	

5.1 Experimental Setup

Experimental environment. Experiments for PHI and PHI-Fuzz evaluation were conducted in an Intel® Core™ i7-8700 CPU @ 3.20GHz, 8GB RAM, and Ubuntu 18.04.4 LTS (VM) environment.

Experiment data.

Table 3 presents the information on the firmware used to evaluate PHI and PHI-Fuzz. The firmware was based on STM32F469NI and STM32F103RB, with the source code collected from an open-source project on GitHub and then ported for use. The per firmware included GPIO, UART, I2C, and SPI for evaluation. In total, four HAL-based firmware and ten pseudo-HAL-based firmware were created and used for the experiments. Additionally, one HALucinator benchmark firmware and two P^{2}IM benchmark firmware were used in the experiment. The firmware was compiled without optimization using the 2018_q4 (gcc8) version. The HAL object file required for Libmatch, a program that compares with PHI, was also compiled with the 2018_q4 (gcc8) version and without optimization.

Table 4. PHI result of UART_Hyperterminal_IT by Optimization level

Optimization level	Total(#)	Result(%)
`-O0` No optimization	16	69
`-O1` Reduced code size, execution time	15	75
`-O3` Optimization of inline functions and registers	15	75
`-Os` Omit optimizations that increase code size	15	75
`-Og` Remove optimizations that confuse debugging	15	75

5.2 Scalability of PHI (RQ1)

To demonstrate PHI's scalability, this study shows that identifying pseudo-HAL functions is feasible with only the MCU name, without relying on detailed firmware information. To validate this claim, function identification experiments were conducted on compiled firmware at various optimization levels, and the function identification rates were compared with LibMatch's HAL function identify results when the compiler versions of the SDK file and the target firmware differed. The reason for demonstrating scalability through results obtained with different compilation options is that LibMatch, which uses the specific SDK, exhibits varying results depending on compilation options, as shown in Figure 2. Therefore, by achieving consistent results without using the SDK, PHI establishes its scalability. Table 4 presents the PHI results for the UART_Hyperterminal_IT [11] firmware compiled at different optimization levels using the same source code. Optimization led to a reduction of one in the total number of peripheral-related functions (HAL functions), but at all optimization levels, 15 identical pseudo-HAL function identifications were possible. In comparison, LibMatch's identification rate varies depending on the compilation level of the SDK and firmware, unlike PHI, which not only requires the SDK but also shows consistent identification results in target firmware compiled at each optimization level.

As an additional experiment, a comparison experiment was conducted by detecting with a different compiler. While the original experimental firmware and SDK files were compiled with 2018_q4 (gcc8), for this experiment, only the experimental firmware was compiled with 2016_q4 (gcc6) to compare the results in the unideal environment. Figure 5 and 6 show the results of PHI and LibMatch with four types of firmware that utilize HAL functions and compiled with 2018_q4 (gcc8) and 2016_q4 (gcc6) each. Figure 5 represents the identification results in an ideal environment for using LibMatch. As a result, PHI exhibited an average exploration rate of around 69%, while LibMatch showed an average exploration rate of approximately 75%. Figure 6 illustrates the results of experiments conducted using firmware compiled with 2016_q4 (gcc6), which did not occur in an ideal environment. PHI, since it doesn't rely on the SDK, produced the same results as the exploration with the firmware compiled with 2018_q4 (gcc8). However, LibMatch did not achieve the same results. LibMatch detected only NVIC-related functions, resulting in detection performance of up to

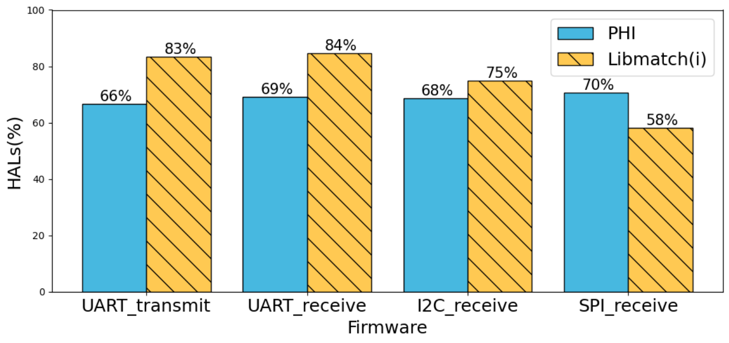


Fig. 5. Comparison of HAL function identification rates between PHI and LibMatch. The figure shows the execution outcome of the LibMatch with the ideal compiler version.

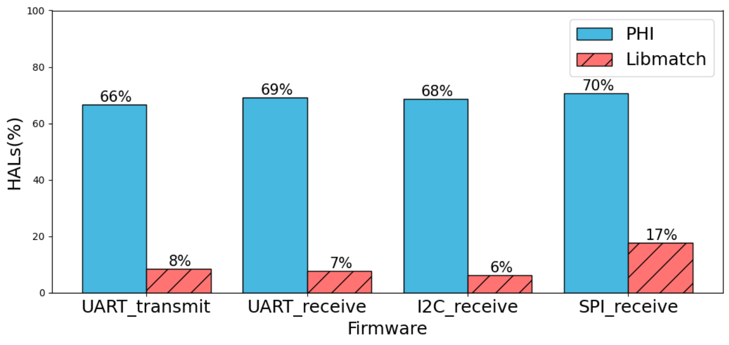


Fig. 6. Comparison of HAL function identification rates between PHI and LibMatch. The figure shows the execution outcome of the LibMatch without the ideal compiler version.

17% or less. As a result of these experiments, it was confirmed that PHI can explore functions consistently across various compilation optimization options and compiler versions, demonstrating its scalability as a program. With this scalable feature of PHI, it is possible to detect peripheral-related functions in commercially available firmware without prior information. These detection results can subsequently be used for vulnerability exploration through PHI-Fuzz. The experimental results related to this will be presented in Section 5.4.

5.3 Effectiveness of PHI (RQ2)

In Section 5.2, it was observed that LibMatch's identification rate is favorable when the SDK is in an ideal environment. Therefore, in this section, we compare LibMatch and our approach in the ideal environment. Generally, the HAL function identification rate of PHI closely resembled LibMatch's

Table 5. HAL function identification result for SPI_receive firmware

Function	Libmatch	PHI
`HAL_GPIO_Init`		✓
`HAL_NVIC_SetPriority`	✓	
`HAL_NVIC_SetPriorityGroup`	✓	
`HAL_RCC_ClockConfig`	✓	✓
`HAL_RCC_GetHCLKFreq`	✓	
`HAL_RCC_GetPCLK1Freq`	✓	✓
`HAL_RCC_GetPCLK2Freq`	✓	✓
`HAL_RCC_GetSysClockFreq`	✓	✓
`HAL_RCC_OscConfig`	✓	✓
`HAL_SPI_Init`		✓
`HAL_SPI_MspInit`		
`HAL_SPI_Receive`		✓
`HAL_SPI_Transmit`		✓
`HAL_SPI_TransmitReceive`		✓
`HAL_UART_Init`		✓
`HAL_UART_MspInit`	✓	
`HAL_UART_Transmit`	✓	✓
Total	10	12

rate (as shown in Figure 5). However, for UART_transmit, UART_receive, and I2C_receive firmware, LibMatch displayed a higher search rate than PHI. What could be the reason? The functions identified by LibMatch but not by PHI were NVIC-related functions, specifically *HAL_NVIC_SetPriority* and *HAL_NVI C_SetPriorityGrouping*. PHI failed to identify these functions because the NVIC-related DB configuration was not established in PHI since the access address was outside the MMIO range. Conversely, for SPI_receive firmware, PHI exhibited a higher search rate than LibMatch. In Table 5, while PHI did not identify two NVIC-related functions, LibMatch could not identify four other SPI-related functions. This confirms that LibMatch cannot identify all HAL functions, whereas PHI can identify functions that LibMatch cannot.

Additionally, Table 6 shows the results of another function identification experiment using 10 firmware that call pseudo-HAL functions instead of HAL functions. While LibMatch had a detection rate of 0%, PHI could identify functions at a significantly high rate of 92.3%. As a result, PHI can identify HAL functions with performance similar to or even superior to LibMatch, which has access to all SDK information, even without utilizing the SDK. Additionally, PHI can also identify pseudo-HAL functions that were previously inaccessible for exploration with LibMatch. Furthermore, similar to the results in Section 5.2 , PHI's effectiveness in detecting a wider range of peripheral-related functions allows for more efficient fuzzing, making it beneficial.

Table 6. Pseudo-HAL function identification(%)

Firmware	Libmatch	PHI
Baremetal_I2C	0	68.1
FreeRTOS_I2C	0	63.6
Baremetal_UART	0	64.2
FreeRTOS_UART	0	64.2
RIOT_I2C_receive	0	60
RIOT_I2C_transmit	0	62.5
RIOT_SPI_receive	0	60
RIOT_UART	0	64.7
RIOT_SPI	0	92.3
RIOT_I2C	0	84.2

Table 7. Fuzzing experiment

Firmware	HAL-Fuzz	PHI-Fuzz
UART_receive	O	O
I2C_receive	O	O
UART_HyperTerminal_IT	O	O
Drone	O	O
CNC	X	O
Baremetal_I2C	X	O
FreeRTOS_I2C	X	O
Baremetal_UART	X	O
FreeRTOS_UART	X	O

5.4 Effectiveness of PHI-Fuzz in bug finding (RQ3)

To demonstrate the effectiveness of PHI-Fuzz, the fuzzing results of PHI-Fuzz and HAL-Fuzz were compared. Table 7 represents the results of testing the feasibility of fuzzing on nine firmware, using HAL-Fuzz and PHI-Fuzz. Among the experimental firmware, UART_receive, I2C_receive, UART_HyperTerminal_IT, and Drone contain HAL functions, and both HAL-Fuzz and PHI-Fuzz can be used to fuzz these samples. However, CNC, Baremetal_ I2C, FreeRTOS_I2C, Baremetal_UART, and FreeRTOS_UART contain pseudo-HAL functions, and can only be fuzzed using PHI-Fuzz.

Table 8 shows the execution results of HAL-Fuzz and PHI-Fuzz on Drone and CNC. The experimental results reveal that both fuzzers could run on Drone, but only PHI-Fuzz was capable of running on CNC. PHI-Fuzz outperformed in terms of fuzzing execution speed and execution path on Drone, as more functions were identified and handled. Furthermore, PHI-Fuzz discovered six unique crashes not detected by HAL-Fuzz, indicating that PHI-Fuzz demonstrated superior performance in finding bugs.

Table 8. Fuzzing experiment with Drone and CNC firmware

Firmware	HAL-Fuzz			PHI-Fuzz		
	Exec.	#Path	#Crash	Exec.	#Path	#Crash
Drone	2,981,648	473	✗	3,511,621	491	✗
CNC	✗	✗	✗	4,020,289	958	6

Table 9. Drone firmware fuzzing Performance Comparison in terms of execution speed & a number of basic blocks.

	HAL-Fuzz [3]	P^2IM [12]	HEFF [15]	Fuzzware [20]	PHI-Fuzz
Modeling level	Function	Register	Dual	Register	Function
Function scalable	HAL	HAL Pseudo-HAL	HAL Pseudo-HAL	HAL Pseudo-HAL	HAL Pseudo-HAL
Speed(exec/s)	49	20	21	23	53
Executed BB (#)	254	519	707	377	210

6 Discussion & Limitation

The results presented in Section 5.2 demonstrate that PHI can effectively identify both pseudo-HAL and HAL functions independently of firmware information, as shown in Section 5.3. Moreover, due to its scalability, PHI can efficiently find bugs, as discussed in Section 5.4. Furthermore, the HAL function identification results in TABLE 5 reveal that PHI outperforms LIBMATCH, since it identified four out of the five SPI-related functions that LIBMATCH failed to identify. However, LIBMATCH has not yet identified *HAL_RCC_GetHCLKFreq* and *HAL_UART_MspInit*. Therefore, to achieve high function coverage during fuzzing, a dual identification technique can be employed. This approach involves first identifying function information through LIBMATCH and then executing PHI to identify functions related to all peripheral devices within the MMIO range.

TABLE 9 compares the fuzzing performance of firmware fuzzers at various levels. As seen in the table, PHI-Fuzz exhibits more than twice the speed compared to register-level fuzzers and is 8% faster than the function-level firmware fuzzer HAL-FUZZ, achieving the best results in terms of fuzzing speed. However, it also obtained the lowest number of executed basic blocks. This is because register-level firmware fuzzers process all registers, resulting in a larger number of executed basic blocks. On the other hand, function-level firmware fuzzers execute a relatively smaller number of basic blocks since they have predefined handlers for each function call. In this context, PHI explored and handled more functions than HAL-FUZZ, leading to the execution of the fewest basic blocks.

7 Related Work

Firmware fuzzing for an MCU target requires firmware emulation. Unlike general software, firmware depends on various peripheral devices, making peripheral device emulation the core of firmware emulation. To address this dependency problem of peripheral devices, various firmware emulation studies have been conducted. In this section, we introduce the firmware emulation technique and the latest fuzzers that utilize it.

7.1 Firmware Emulation

In WYCINWYC [19], firmware emulation is divided into two categories: full emulation, which emulates both the core and peripheral devices of the firmware, and partial emulation, which emulates only the core device and handles peripheral device emulation through physical hardware or peripheral modeling. Full emulation requires significant engineering effort, as all peripherals must be directly configured into the emulator. In particular, in the case of MCUs, which can have various manufacturers and peripheral devices, directly emulating all of them incurs high costs. On the other hand, partial emulation is proposed to mitigate the inefficient development effort of peripheral devices required during full emulation. This method was studied using hardware-in-the-loop (HITL) and peripheral modeling techniques.

The hardware-in-the-loop emulation handles peripheral access by using real peripheral hardware [17, 23]. This approach performs firmware emulation by communicating with peripherals not supported by the emulator using actual peripheral hardware. However, its availability is limited due to the requirement of actual peripheral hardware. On the other hand, peripheral modeling emulates I/O processing for peripheral devices through a model of the peripheral device [10–13, 26]. This method does not use actual peripheral devices, making it easier to use and reducing engineering efforts. Muench et al. [19] demonstrated that emulation through peripheral modeling is more effective than the HITL method and improves emulation performance.

7.2 Hardware-Level Emulation

Peripheral modeling can be categorized into hardware-level, function-level, and register-level modeling based on the modeling level of the peripheral device. Pretender [13] models a peripheral device based on hardware values obtained by inputting values for the actual device. The modeling process uses machine learning, and firmware fuzzing is performed using the implemented model. This is different from the HITL method in that the hardware is used only during the peripheral modeling phase. Thus, fuzzing can proceed without an actual device, relying solely on the modeled result. However, a drawback of this approach is that various hardware is eventually required for the peripheral modeling phase. In contrast, PHI makes it possible to identify functions related to peripheral devices using only firmware binary images and MCU names, without the need for actual hardware at any stage. This enables more scalable fuzzing than Pretender and other peripheral modeling-based approaches.

7.3 Function-Level Emulation

HALucinator [11] is an emulator that allows developers to model peripheral devices of MCU devices directly using the Hardware Abstraction Layer (HAL). Compared to full emulation, which requires detailed modeling of the register unit, HALucinator reduces overhead by allowing developers to directly model the HAL, which is commonly used in many MCU target operating systems. When HAL functions are called, HALucinator handles them by using modeled function handlers. Moreover, HALucinator provides emulation for each peripheral device in the HAL layer, making it possible to fuzz without emulating complex hardware. PHI-Fuzz uses a self-modified HAL-Fuzz function handler for fuzzing. Furthermore, PHI's ability to identify pseudo-HAL functions addresses the limitation of HALucinator, which could only identify HAL functions.

7.4 Register-Level Emulation

Compared to HALucinator, which focuses on handling functions, P2IM [12] is designed for dynamic testing and fuzzing of individual I/O devices at the register level. When the firmware is executed in the emulator, P2IM classifies the access pattern of the peripheral's MMIO registers into categories such as `CR, SR, DR`, and `C&SR` using a proposed heuristic and performs peripheral device modeling with each register handling method. As a result, P2IM does not require prior knowledge of which specific peripheral devices are connected to the MCU since peripheral device handling is performed automatically. PHI leverages P2IM's register access pattern classification to identify peripheral functions. By analyzing the MMIO information output through DDG, PHI classifies peripherals and calculates the used offset, categorizing them into memories such as `SR, DR,` and `CR`. Through this classification process, PHI identifies the accesses performed by the HAL and pseudo-HAL functions. In contrast to P2IM, which automatically creates and operates a handler during fuzzing, PHI-Fuzz requires only a pre-written function handler for the identified function, enabling faster fuzzing.

Laelaps [10] performed firmware emulation through dynamic symbolic execution when an undefined peripheral device access occurred in the emulator while being emulated through QEMU. µEmu [26] analyzed register access patterns for peripheral access via symbolic execution, prior to firmware fuzzing. During symbolic execution, rules for responding to unknown peripheral accesses are inferred, stored in the Knowledge Base (KB), and referenced in the firmware analysis. To address the limitations of Laelaps and µEmu, Fuzzware [20] proposes a solution for limiting fuzzing coverage expansion through path removal during symbolic execution and partial input overhead. PHI also leverages symbolic execution to extract the called functions. Function identification information is provided through Angr, a symbolic execution tool. The offset used when the address of the called function is in the MMIO range is extracted, and function matching is performed through this information.

8 Conclusion

This study aims to improve firmware fuzzing efficiency by identifying both HAL and pseudo-HAL functions at the register level and implementing PHI and PHI-Fuzz as firmware fuzzers based on HAL-Fuzz. The proposed method was able to identify HAL functions accessing the MMIO range at a comparable level to LibMatch, while also addressing the limitation of LibMatch in identifying pseudo-HAL functions. PHI-Fuzz proved to be more effective in bug finding than HAL-Fuzz, as it discovered additional crashes not found by HAL-Fuzz. However, there are still some functions that LibMatch can identify but PHI cannot. To address this, future work will involve conducting a study that combines LibMatch and PHI to increase the function identification rate.

Acknowledgement. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (RS-2023-00229400, Development of user authentication and privacy preserving technology for a secure metaverse environment) and by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2023-00230337, Advanced and Proactive AI Platform Research and Development Against Malicious Deepfakes)

References

1. State of IoT_spring-2023: `https://iot-analytics.com/product/state-of-iot-spring-2023/`
2. National vulnerability database. Accessed: Jun. 1, 2021. [Online]. Available: `https://nvd.nist.gov/vuln/`
3. HAL_Fuzz. Accessed: Jun. 1, 2021. [Online]. Available: `https://github.com/ucsb-seclab/hal-fuzz`
4. Libmatch. Accessed: Jun. 1, 2021. [Online]. Available: `https://github.com/subwire/libmatch`
5. 2019 Embedded markets study.(2019) Accessed: Jun. 1, 2021. [Online]. Available: `https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf`
6. Description of STM32F4 HAL and low-layer drivers: `https://www.st.com/resource/en/user_manual/dm00105879-description-of-stm32f4-hal-and-ll-drivers-stmicroelectronics.pdf`
7. P^{2}IM real-world firmware samples. Accessed: Jun. 1, 2021. [Online] Available: `https://github.com/RiS3-Lab/p2im-real_firmware/tree/d4c7456574ce2c2ed038e6f14fea8e3142b3c1f7`
8. Angr : `https://github.com/angr/angr`
9. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proc. USENIX annu. technical conf., FREENIX Track. Berkeley, CA, USA (Apr 2005)
10. Cao, C., Guan, L., Ming, J., Liu, P.: Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In: Annual Computer Security Applications Conference. pp. 746–759 (2020)
11. Clements, A.A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., Payer, M.: Halucinator: Firmware re-hosting through abstraction layer emulation. In: Proc. USENIX Secur. Symp. pp. 1201–1218 (2020)
12. Feng, B., Mera, A., Lu, L.: P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In: Proc. USENIX Secur. Symp. pp. 1237–1254 (2020)
13. Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y.R., Kruegel, C., et al.: Toward the analysis of embedded firmware through automated re-hosting. In: Proc. Int. Symp. Research in Attacks, Intrusions and Defenses (RAID). pp. 135–150. Beijing, China (Sep 2019)
14. He, Y., Zou, Z., Sun, K., Liu, Z., Xu, K., Wang, Q., Shen, C., Wang, Z., Li, Q.: Rapidpatch: Firmware hotpatching for real-time embedded devices. In: 31th USENIX Security Symposium (USENIX Security 22) (2022)
15. Hwang, E., Lee, H., Jeong, S., Cho, M., Kwon, T.: Toward fast and scalable firmware fuzzing with dual-level peripheral modeling. IEEE Access **9**, 141790–141799 (2021)
16. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proc. ACM Conf. Comput. Commun. Secur. (CCS). pp. 2123–2138. Toronto, Canada (Oct 2018)
17. Koscher, K., Kohno, T., Molnar, D.: SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In: Proc. USENIX Workshop Offensive Technol. Washington, DC, USA (Aug 2015)

18. Mera, A., Feng, B., Lu, L., Kirda, E., Robertson, W.: DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis. In: Proc. IEEE Symp. Secur. and Privacy (SP). pp. 302–318. Los Alamitos, CA, USA (may 2021)
19. Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D.: What You Corrupt Is Not What You Crash: Challenges in fuzzing embedded devices. In: Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS). San Diego, CA, USA (January 2018)
20. Scharnowski, T., Bars, N., Schloegel, M., Gustafson, E., Muench, M., Vigna, G., Kruegel, C., Holz, T., Abbasi, A.: Fuzzware: Using precise mmio modeling for effective firmware fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 1239–1256 (2022)
21. Spensky, C., Machiry, A., Redini, N., Unger, C., Foster, G., Blasband, E., Okhravi, H., Kruegel, C., Vigna, G.: Conware: Automated modeling of hardware peripherals. In: Proc. ACM Asia Conf. on Comput. Commun. Secur. (Asia CCS). pp. 95–109 (2021)
22. Wright, C., Moeglein, W.A., Bagchi, S., Kulkarni, M., Clements, A.A.: Challenges in firmware re-hosting, emulation, and analysis. J. ACM Computing Surveys (CSUR) **54**(1), 1–36 (2021)
23. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al.: AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In: Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS). vol. 23, pp. 1–16. San Diego, CA, USA (February 2014)
24. Zalewski, M.: American funzz lop. Accessed: Jun. 1, 2021. [Online].Available: `https://lcamtuf.coredump.cx/afl/`
25. Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L.: FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: Proc. USENIX Secur. Symp. pp. 1099–1114. Santa Clara, CA, USA (August 2019)
26. Zhou, W., Guan, L., Liu, P., Zhang, Y.: Automatic firmware emulation through invalidity-guided knowledge inference. In: Proc. USENIX Secur. Symp. (2021)