IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 29, 2021, accepted September 25, 2021, date of publication October 8, 2021, date of current version October 25, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3118854

Toward Fast and Scalable Firmware Fuzzing With Dual-Level Peripheral Modeling

EUNBI HWANG, HYUNSEOK LEE, SEYEON JEONG, MINGI CHO, AND TAEKYOUNG KWON, (Member, IEEE)
Graduate School of Information, Yonsei University, Seoul 03722, South Korea

Corresponding author: Taekyoung Kwon (taekyoung@yonsei.ac.kr)

This work was supported by the Defense Acquisition Program Administration and Agency for Defense Development under Contract UD190016ED.

ABSTRACT Firmware vulnerabilities raise serious security concerns with the rapid growth in connected embedded devices. Fuzzing is an effective dynamic testing technique to find those vulnerabilities; however, firmware fuzzing is very limited by hardware dependence, such as on-chip and off-chip peripherals. The latest elegant approaches are making substantial progress in hardware-independent firmware fuzzing, but there is room for further improvement. We observe that hardware-independent peripheral modeling is scalable but slow at the register level; in contrast, at the abstract function level, it is fast but has limited scalability. Firmware fuzzing is still challenging in terms of achieving both scalability and efficiency. To address this problem, we present a dual-level approach that leverages register level modeling and selective function level modeling in a hybrid manner. Our method starts firmware fuzzing at the register level and connects peripheral handlers while executing hardware abstraction layer functions. We evaluate our method in terms of efficiency, scalability, and effectiveness with four real-world firmware and demonstrate the possibility of relatively fast and scalable firmware fuzzing that combines the benefits of the two levels.

INDEX TERMS Firmware, fuzzing, peripheral modeling, security, vulnerability.

I. INTRODUCTION

Embedded systems are prevalent in our daily lives, including but not limited to the Internet of Things (IoT), automation systems, and smart cars. In particular, the growth in connected IoT devices is spectacular; for example, the revenue of such devices was approximately 12 billion in 2020 (about 15 times compared to 2010), which is likely to exceed 30 billion in 2025 [1]. This rapid progress has resulted in a significant increase in firmware vulnerabilities, which are listed in the NIST National Vulnerability Database [2]. Firmware vulnerabilities are critical security concerns, leading to not only digital but also physical damage through substantial failures, such as (late) crashes, reboots, and hangs, and even silent faults, in embedded systems [3], [4]. They present a much higher impact in general terms than the ones in OS or application-level vulnerabilities because they are at the low level of the infrastructure [5]. Unfortunately, however, it is cumbersome and difficult to test firmware from a security perspective, particularly when running on microcontrollers called MCUs and even worse as closed-source and blob binaries, due to the complex environmental dependencies required at run time [6], [7].

Fuzzing is an effective and scalable testing method that is widely used to detect vulnerabilities in modern software [8]. Coverage-guided fuzzers, such as AFL [9], have successfully discovered many bugs in a variety of software [10]. Thus, the application of coverage-guided fuzzing to MCU firmware is highly desirable; however, this task is challenging [4]. Although on-chip fuzzing is preferable to address hardware dependencies, it is infeasible because of the limited resources in MCUs. Hardware emulation may deal with this problem by running every element on commodity computers, but this task is laborious and impractical because of the notorious heterogeneity in embedded hardware. For example, both on-chip/off-chip peripherals and architectural derivatives are extremely diverse in this domain. Indeed, none of the modern emulators can emulate the whole range of MCU peripherals, entailing that firmware executions are disrupted.

Recently, Muench *et al.* [4] studied firmware fuzzing and suggested that partial emulation with peripheral modeling might be a reasonable direction to the hardware-independent

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek.

(peripheral-agnostic) firmware fuzzing. As a quick response to this, two invaluable methods were proposed. Feng *et al.* [7] presented P^{2}IM, which abstracts diverse peripherals at the memory-mapped register level and handles firmware I/O based on automatically generated models for fuzzing. Clements *et al.* [6] proposed an approach of peripheral modeling at the abstract function level by replacing hardware abstraction layer (HAL) functions with software handlers for fuzzing; they introduced HAL-Fuzz along with a high-level emulation system called HALucinator. Although both of these approaches realize hardware-independent firmware fuzzing, there remains a scope for further improvement. We observe that P^{2}IM is scalable but relatively slow, whereas HAL-Fuzz is fast but less scalable and requires HAL functions and exact matching environments. Thus, it is a challenging task to satisfy both the scalability and efficiency of firmware fuzzing.

In this paper, we address this problem by presenting a dual-level approach that leverages both register level peripheral modeling and selective function level peripheral modeling in a hybrid manner. Our prototype system called hybrid emulation for firmware fuzzing (HEFF), starts firmware fuzzing at the register level and connects peripheral handlers while executing HAL functions. We combine the benefits of the two levels for fast and scalable firmware fuzzing. We implement our system based on P^{2}IM and HAL-Fuzz, and perform evaluation in terms of efficiency, scalability, and effectiveness.

A. CONTRIBUTION

This paper makes the following contributions.

- **Dual-Level Peripheral Modeling.** We proposed a novel approach of peripheral modeling in a hybrid manner to improve hardware-independent firmware fuzzing.
- **Fast and scalable firmware fuzzing.** We implement our prototype system and conduct fuzzing experiments.
- **Effectiveness of bug finding.** Our prototype system finds more bugs and finds them faster.

B. ORGANIZATION

Section II reviews related work. Section III describes backgrounds of firmware fuzzing and conducts small experiments for our motivation. Section IV represents our system design. Section V reports our evaluation. Section VI makes a discussion. Section VII concludes this paper.

II. RELATED WORK

We classify the previous work related to this study into three categories. Readers are referred to the WYCINWYC paper [4] for more about firmware fuzzing and its challenges.

A. FULL EMULATION

In general-purpose OS-based devices, since hardware interaction occurs in the kernel, it is possible to process these interactions through the emulated kernel without special emulations of the peripherals. Firmadyne [11], Costin *et al.* [12], and FirmFuzz [13] are some of the reported vulnerability-finding studies targeting Linux-based embedded devices; these are based on a full system emulation of QEMU [14]. Additionally, FIRM-AFL [10] is an augmented process emulation technique that combines QEMU's full system-mode and user-mode emulations to improve the compatibility and fuzzing performance.

Embedded-OS or no-OS abstraction-based devices communicate directly with the peripherals such that firmware emulation requires peripheral emulations. Moreover, all documents are required for full emulation of the board and peripherals of the target device. QEMU_STM32 [15] performs emulations using the STM32 library based on the expanded QEMU. Panda [16] uses the register knowledge of each board through the SDK and performs emulations by implementing all the logic for peripheral operation at the register-level in QEMU.

The accuracy of system-vulnerability detection through full emulation of the firmware is high, however, detailed knowledge of the hardware is often lacking, and even with sufficient information, a significant amount of engineering effort required to implement full emulators [4], [17]. Moreover, there is a disadvantage that such emulations are slow [4]. Therefore, HEFF, which explores vulnerabilities targeting embedded-OS or no-OS abstraction-based devices, uses a partial emulation technique that overcomes this limitation of full emulation.

B. PARTIAL EMULATION (HARDWARE-IN-THE-LOOP)

Hardware-in-the-loop partial emulations require firmware images and physical devices. Avatar [18] solves the hardware resource problem and performs firmware emulation by forwarding the state changes of all the memory and CPU registers of the emulator to the actual device. PROSPECT [19] provides a proxy that and tunnel arbitrary peripheral accesses within the virtual machine to the embedded system under test, while SURROGATES [20] provides an FPGA bridge between the host and target to reduce the operational overheads and enable analyses of complex systems. Further, Avatar^2 [21] is a multi-target orchestration platform that is implemented by extending Avatar. This approach enables state transfer between tools such as GDB, QEMU, angr, OpenOCD, and PANDA; WYCINWYC [4], PRETENDER [22], Unicorefuzz [23], and HALucinator [6] use Avatar^2 between the emulators or analysis tools for firmware operation.

This method overcomes the limitation of full emulation in that it reduces the engineering effort in implementation by emulating physical devices. However, as described in §I, it is not easy to provide physical devices for emulation because there are various peripherals used in the embedded device. HEFF eliminates these peripheral dependencies that exist in hardware-in-the-loop by using the modeled peripheral instead of the physical device during partial emulation.

C. PARTIAL EMULATION (PERIPHERAL MODELING)

Partial emulations with peripheral modeling require firmware images and peripheral modeling without the firmware source code or hardware. PRETENDER [22] performs peripheral modeling using machine-learning techniques, and P^{2}IM [7] processes the peripheral I/O by abstracting them at the register-level. However, HALucinator [6] solves the peripheral device problem of firmware emulation by partial emulation using the processing method as a peripheral abstracted handler at the HAL function-level. HAL-Fuzz [24] performs firmware fuzzing by combining HALucinator and AFL. In addition, a recent study showed that, DICE [25] and Conware [26] can improve the limitations of P^{2}IM [7] by modeling DMA or using symbolic execution.

Unlike these approaches, HEFF combines register- and function-level modeling; hence, we compared HEFF with P^{2}IM and HAL-Fuzz and observed that our dual-level peripheral modeling in emulation was scalable and efficient.

III. MOTIVATION

This section briefly describes the background of our study and clarifies the research problem through small experiments.

A. BACKGROUND

1) FIRMWARE IN EMBEDDED DEVICES

Firmware is a low-level software that provides hardware controls in various embedded devices. Muench *et al.* [4] categorized those embedded devices into the following three classes according to the type of an operating system: general purpose OS, embedded OS, and no OS abstraction. At present, approximately 81% of the embedded devices are working with MCU firmware with no or less OS abstraction [27], which is our major concern in this study.

2) FIRMWARE FUZZING

Fuzzing is a dynamic testing technique to discover hidden bugs, mainly in software, through random garbled inputs. Literally speaking, fuzzing embedded systems is similar to walking barefoot on a rough gravel road. Instrumentation tools on firmware without emulation are usually not possible. Full emulation of both processor and peripheral devices is impractical because of hardware diversity; furthermore, partial ''hardware-in-the-loop'' emulation also presents poor fuzzing performance due to the bottleneck in communication with actual hardware devices. Such emulation is usually *slow* and *unscalable* for firmware fuzzing.

As explored in [4], *partial emulation with peripheral modeling* is a promising approach to firmware fuzzing, in which a processor is fully emulated while peripherals are only virtually modeled in software. This concept is convincing as any firmware can run on an emulator without real or fully emulated peripherals, as long as the emulator can provide acceptable peripheral data to firmware whenever required. Moreover, it is sufficient to consider only on-chip peripherals in fuzzing because firmware cannot access off-chip peripherals directly. Recently, we observed two such promising approaches: P^{2}IM [7] and HAL-Fuzz [6], [24].

TABLE 1. Fuzzing performance of P^{2}IM and HAL-Fuzz on Robot firmware (5h).

Firmware	Speed (run/s)		Basic blocks	
	P^{2}IM	HAL-FUZZ	P^{2}IM	HAL-FUZZ
Robot	22.23	97.68	1166	315

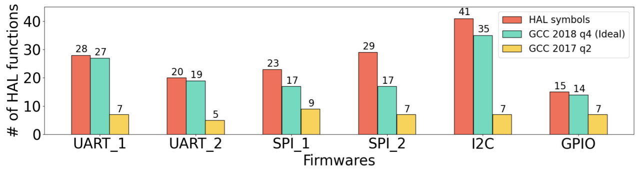


FIGURE 1. Libmatch's HAL function matching results.

B. PROBLEM DEFINITION

Our fundamental question is whether the partial emulation with peripheral modeling is fast and scalable for fuzzing. Therefore, we conducted two small experiments to clarify the research problem of our study.

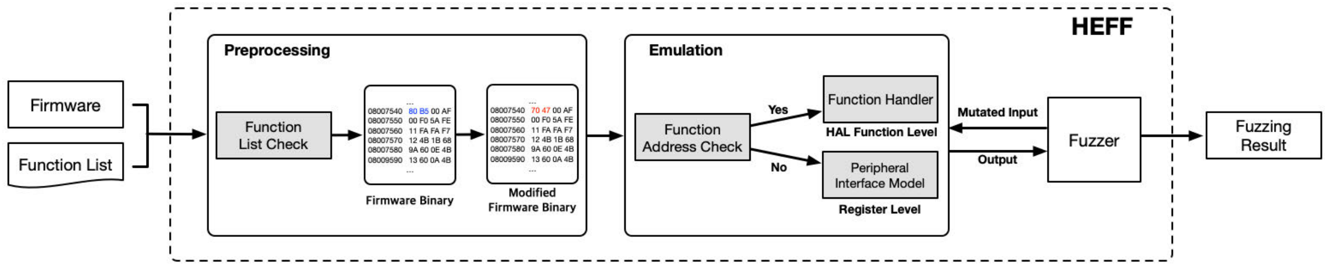
1) REGISTER-LEVEL PERIPHERAL MODELING

P^{2}IM [7] treats peripherals as a black box by manipulating memory-mapped registers reserved for peripherals, rather than emulating any peripherals, so as to provide the equivalent processor-peripheral interfaces to a generic emulator (such as QEMU) for firmware fuzzing. For example, on ARM Cortex-M MCUs, peripheral registers are mandatorily mapped to the `0x40000000-0x5fffffff` memory segment; thus, P^{2}IM considers each memory word in this segment as a potential memory-mapped register. To this end, P^{2}IM automatically instantiates the interface models for firmware specific information; it also monitors and handles every access to the peripheral registers. Although P^{2}IM is highly scalable for firmware fuzzing, we observed that the performance at the register-level can be further improved in terms of speed.

Table 1 shows the result of our experiment that compares P^{2}IM and HAL-Fuzz in terms of fuzzing performance (runs per second) on a firmware (960KB) that includes HAL functions. Although the number of basic blocks executed by P^{2}IM is more than HAL-Fuzz which performs fuzzing through an external handler, P^{2}IM is about four times slower than HAL-Fuzz for fuzzing. Readers are referred to Table 3 for further comparisons.

2) FUNCTION-LEVEL PERIPHERAL MODELING

HAL-Fuzz [6], [24] provides high-level replacements for HAL functions located by library matching techniques in firmware. It handles external interactions between the emulated firmware and the corresponding peripheral models for



```
...
08007540  80 B5 00 AF
08007550  00 F0 5A FE
08007560  11 FA FA F7
08007570  12 4B 1B 68
08007580  9A 60 0E 4B
08009590  13 60 0A 4B
...
```

```
...
08007540  70 47 00 AF
08007550  00 F0 5A FE
08007560  11 FA FA F7
08007570  12 4B 1B 68
08007580  9A 60 0E 4B
08009590  13 60 0A 4B
...
```

FIGURE 2. HEFF System Overview. In the emulation, whether the function which is in the function list is hooked in the firmware is expressed as "Yes" or "No."

fuzzing without any peripheral emulation. HAL-Fuzz needs a specific tool named LibMatch to locate HAL functions in stripped binary firmware, which is highly dependent on the HAL object files that are used to identify HAL functions from arbitrary firmware in LibMatch. Although HAL-Fuzz is fast for firmware fuzzing thanks to the lower granularity at the abstract function-level, we observed that rigorous requirements for the library matching environments can be relaxed to further improve the scalability.

Figure 1 illustrates the result of our experiment that compares the HAL functions found by LibMatch in the different compilation environments for each firmware. The HAL object files used in our experiment were compiled using GCC (2018 q4 version) with the *-O0* optimization level. For the ideal result of LibMatch, we compiled six bare-metal firmwares which are available in [28] (UART_1 [29], UART_2 [30], SPI_1 [31], SPI_2 [32], I2C [33] and GPIO [34]) in the same environment as that of HAL object files. Moreover, to compare the matching results according to the GCC version, we compiled the firmware using the GCC with *-O0* (2017 q2 version), respectively. Finally, to confirm the results according to the optimization options, we also used GCC (2018 q4) with other optimization levels (*-O1, -O2, -O3, -Os* and *-Og*).

In the ideal case, 95% (median) HAL functions were successfully identified. However, when the compiler versions were different, the matching result became considerably lower. Furthermore, when the optimization levels were even slightly different, no matching result came out. Since HAL-Fuzz is a function hooking-based approach, the exact identification of HAL functions by LibMatch is mandatory for firmware fuzzing and also for the proper emulation.

C. OUR DIRECTION

To achieve the benefits of peripheral modeling at both levels, we present a dual-level approach for peripheral modeling. We implemented a firmware fuzzer based on P^{2}IM and HAL-Fuzz; named HEFF that might enjoy both levels optimistically half and half. In HEFF, the peripheral modeling starts at the register-level, as in P^{2}IM, because it is automatic and scalable. When a HAL function call is detected, function-level peripheral modeling is launched by connecting a peripheral handler, as in HAL-Fuzz. Although constructing the peripheral handler requires an engineering effort, this function-level process accelerates fuzzing. Furthermore, HEFF implemented in this way enable fast fuzzing of firmware which calls both HAL and non-HAL functions (*e.g.,* CNC), which was impossible with existing function-level processing alone, through dual-level peripheral modeling. In addition, we observed that even with fewer handlers, HEFF is faster than P^{2}IM (Table 4) and more scalable than HAL-Fuzz (Table 6). HEFF performs efficient and scalable firmware fuzzing.

IV. SYSTEM DESIGN

A. OVERVIEW

Figure 2 illustrates an overview of HEFF's system architecture. HEFF takes a target firmware and a function list as inputs; after the preprocessing, it initiates firmware emulation via QEMU. Preprocessing enables independent control of functions to be processed at the function level during emulation via binary rewriting. For peripherals that cannot be emulated using QEMU, HEFF's dual-level peripheral modeling is employed. Dual-level modeling commences at register-level first, when the HAL function is hooked through the function address check in real time, peripheral modeling is performed through a handler at the HAL function-level. When the handler processing of the corresponding function is completed, the process returns to the next address of the instruction at which the HAL function is called, and emulation continues at the register-level. Finally, fuzzing in HEFF is performed by transferring the mutated input generated in the AFL to the part that requires an external input among the peripheral models generated at each level.

B. DUAL-LEVEL PERIPHERAL MODELING

1) PREPROCESSING

HEFF's dual-level peripheral modeling requires a pre-built QEMU and modified firmware.

Pre-built QEMU is built by inserting the HAL function list and the corresponding handlers into the QEMU implemented by P^{2}IM. The HAL function list is composed of the function name and its start address, which can be created through the symbol parsing program or LibMatch analysis of the HALucinator [6]. In addition, the function handlers are

Algorithm 1 HEFF Process

```
Input: Firmware binary, Function List
Output: Fuzzing result
F = Firmware binary
$F_{mod}$ = Modified firmware binary
FL = Function list file
$FL_{func}$ = Functions in FL
$FL_{addr}$ = Start address of $FL_{func}$
pc = Program pointer
1: Initiate Function List Check
2:   while run Fl budget reached do:
3:     $FL_{func}$ = get(FL) // $FL_{func}$ is the address to be modified.
4:     run radare2 with $FL_{func}$:
5:       4-byte hex value after $FL_{addr}$ $\rightarrow$ overwrite 70 47 (bx lr)
6:     return $F_{mod}$
7: Initiate Function Address Check
8:   while run $F_{mod}$ with HEFF:
9:     if pc = $FL_{addr}$:
10:        Convert to Function-level
11:          AFL mutated input $\rightarrow$ Function Handler
12:          return fuzzing output
13:      else:
14:        Convert to Register-level
15:          AFL mutated input $\rightarrow$ Peripheral Interface Model
16:          return fuzzing output
17:      return fuzzing result
```

implemented in C-code by modeling the external interactions between the emulated firmware and the peripherals of the corresponding target firmware.

Modified firmware is created by rewriting the hex value to the start point of listed functions so that the internal function is not driven. Functions handled at function-level are only run via handlers. However, when the firmware runs in QEMU, it is difficult to control the entire function with only the start address because the translated tiny block of QEMU constantly changes. Therefore, to prevent internal functions from running, as described in lines 1-6 of Algorithm 1 above, HEFF finds the function start addresses in the binary from a given function list check logic and rewrites the first 2 bytes with *bx lr* (little endian: 70 47). In this manner, the function hooked directly parses the address to return in the function handle and specifies *lr* (r14). If the function is called, *bx lr* code is subsequently executed according to the modified *lr* information, so the function is immediately terminated and the next instruction is executed. Through this process, it is possible to reduce unnecessary internal function operations during emulation.

2) FUNCTION ADDRESS CHECK

Function address check is the core of dual-level emulation, given that the function-level modeling is operated only through this process. The function address check logic monitors the program counter (PC) value, which is changed during firmware operation in real time. At this point, if the PC value matches the address in the function list, that function is hooked and converted to the function-level peripheral modeling.

3) FUNCTION-LEVEL PERIPHERAL MODELING

In HEFF, function-level peripheral modeling is executed by connecting the handler corresponding to the hooked HAL function after the function address check. This peripheral abstraction handler was proposed in HALucinator [6], and it is created by converting the HAL function argument into data that the peripheral model can use. This facilitates easy handling of complex interactions between peripherals and processors. In addition, in most cases, because some hardware concepts such as power or clocking do not exist in the emulation, the return values of the handlers do not affect the emulation.

4) REGISTER-LEVEL PERIPHERAL MODELING

Peripheral modeling at the register-level continues as long as the HAL function is not hooked during emulation. In addition, this method uses P^2IM's peripheral interface modeling. First, it classifies the control register (CR), status register (SR), and data register (DR) via register access patterns, and then defines the peripheral interface to be emulated in the target firmware through model abstraction. Next, by handling peripheral I/O with the peripheral interface defined through model instantiation, the running firmware can emulate without peripheral problems. Using the modeling of P^2IM, HEFF continuously emulates at the register-level when it does not operate at the function-level.

C. FUZZER

HEFF performs fuzzing by providing a mutated input in two ways (represented on lines 11 and 15 of Algorithm 1), given that it carries out peripheral modeling at the dual-level. After peripheral modeling at the register-level, HEFF channels the AFL-mutated input into the DR where the data value store. Whereas, after function-level peripheral modeling, the AFL-mutated input is provided to the function handler by parsing the mutated file of the AFL. For example, the HAL function argument of the receive-related function consists of *(peripheral handle, data variable to store input, length of input value)*. The elements of this stored in the r0, r1, and r2 registers, respectively, when the function is called. Concerning the fuzzing input of the handler a data variable to store the input and the length of the input value are required. The handler parses the r1 and r2 registers, splits the input in the mutated file of the AFL as much as the input value length (r2), and stores it in the data variable (r1) to be saved.

V. IMPLEMENTATION

We implemented HEFF on top of P^2IM. HEFF takes a function list and firmware binary as the inputs. Using these inputs, function list check, function address check processes, and corresponding HAL function handlers are implemented by inserting 10 lines of C code in the QEMU. In addition, the firmware binary is automatically modified by our python script which based on radare2 [35]. Table 2 shows an example of modified firmware.

TABLE 2. Example of modified input firmware.

Address	Firmware				Modified Firmware			
...		...				...		
08007540	80	B5	00	AF	70	47	00	AF
08007550	00	F0	5A	FE	00	F0	5A	FE
08007560	11	FA	FA	F7	11	FA	FA	F7
...		...				...		

Concerning the function handler, we obtained the function-level handling idea from HAL function hooking and handling in HAL-Fuzz. We ported the C code function handler to gnu-eclipse-qemu in P^{2}IM so that it could be used in HEFF. There are two major function handlers implemented in HEFF: a receive handler, which receives the input from the fuzzer, and a return-zero handler, which bypasses functions that are not specifically needed in the fuzzing and returns zero to the r0 register. Through the return-zero handler, we control the init- and transmit-related functions, because these functions do not require inputs during a firmware execution and consequently are not important for the fuzzing process. Meanwhile, we did not modify P^{2}IM for register-level peripheral modeling.

VI. EVALUATION

To evaluate HEFF, we set the following research questions:

- RQ1: Is HEFF efficient? (§VI-B)
- RQ2: Is HEFF scalable? (§VI-C)
- RQ3: Is HEFF effective in bug detection? (§VI-D)

A. EXPERIMENTAL SETUP

1) EXPERIMENT DATA

In the experiments, we used four real firmware provided by P^{2}IM [36]. The real firmware are CNC, PLC, Robot and Drone, which use HAL functions. CNC is a grbl milling controller firmware used in 3D printers and laser cutters, PLC (Programmable Logic Controller) is a firmware that controls important processes as part of the sterilizer, Robot is a motion controller firmware used in personal transport devices, and Drone is an autopilot controller firmware for quad-copters. The experiment was conducted by applying a total of 9 handlers, 4 handlers each including duplicates, to CNC, PLC, Robot, and Drone.

2) SEEDS

The executable seed input values are different for each firmware. In the case of P^{2}IM and HEFF, a random seed value is used, whereas HAL-Fuzz requires an input of a specific length for each firmware. Therefore, the input used in HAL-Fuzz was used in P^{2}IM and HEFF only for the same firmware.

3) PLATFORM AND CONFIGURATION

We conducted the experiments on a virtual machine with Intel® Xeon(R) Gold 6134 CPU and 16GB RAM, running 64-bit Ubuntu 18.04 system. In the experiment, we performed real-world firmware fuzzing 10 times [8] for 5h each.

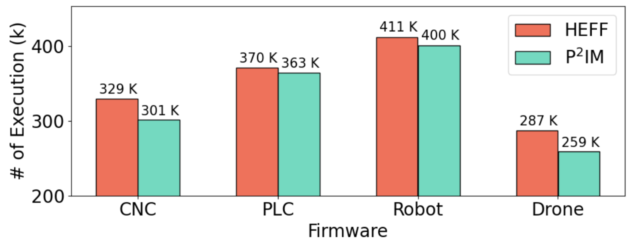


FIGURE 3. The average number of executions of HEFF and P^{2}IM for real firmware on the 10times run (5h).

TABLE 3. Real firmware information.

Firmware	MCU	OS	Size	#HALs
CNC	STM32F429ZI	Bare metal	287KB	54
PLC	STM23F429ZI	Arduino	774KB	54
Robot	STM32F103RB	Bare metal	960KB	43
Drone	STM32F103RB	Bare metal	425KB	41

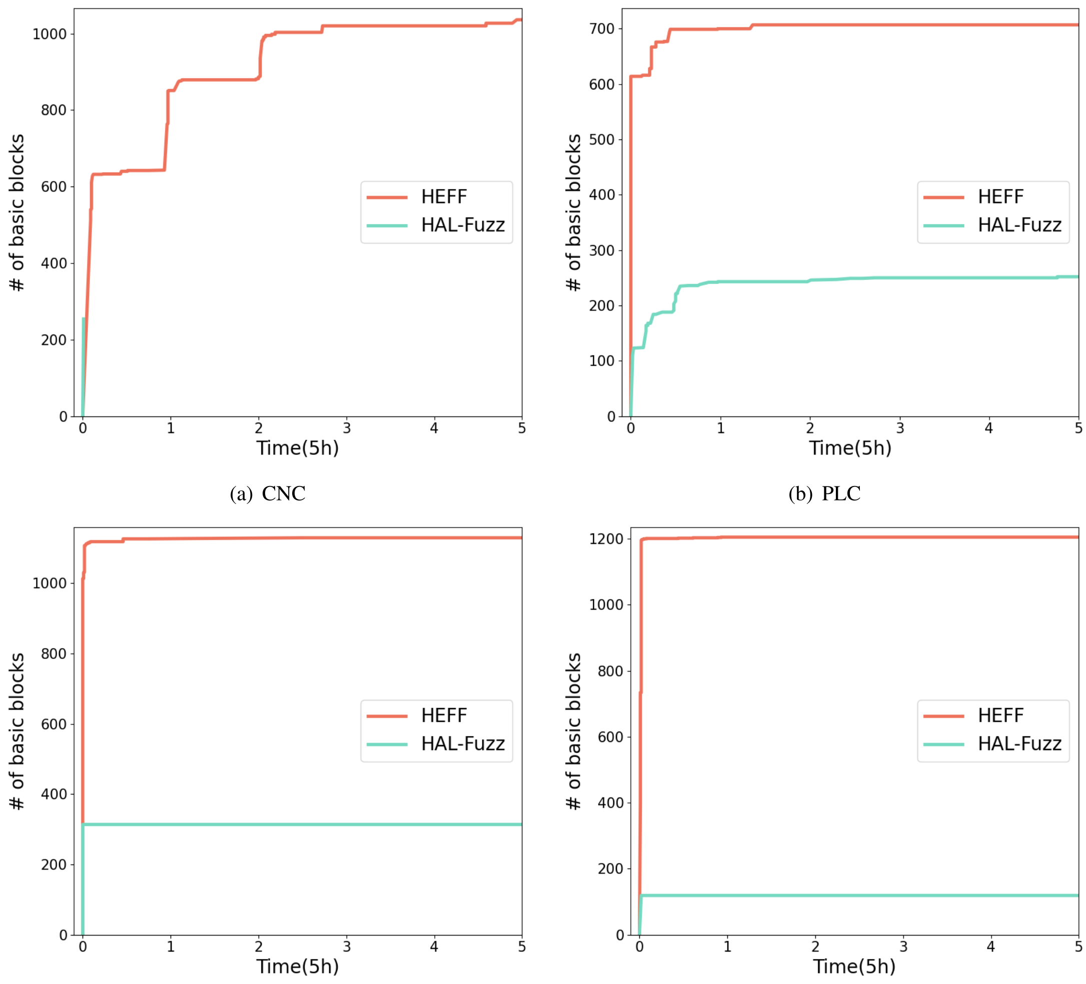
TABLE 4. The number of executions of each firmware at 10 fuzzing run (5h) in HEFF, P^{2}IM, and HAL-Fuzz.

Firmware	HEFF	P^{2}IM		HAL-FUZZ	
	Median	Median	Improv.	Median	Improv.
CNC	329,818	301,151	+8.69%	-	N/A
PLC	370,830	363,210	+2.05%	876,431	-136.34%
Robot	411,730	400,287	+2.27%	1,758,402	-327.07%
Drone	287,528	259,951	+9.59%	530,091	-84.36%

B. RQ.1: EFFICIENCY OF HEFF

To answer RQ1, we compared the fuzzing execution speed with P^{2}IM and HAL-Fuzz to confirm that the function handlers perform fuzzing efficiently. In fuzz testing, faster fuzzers can run more through more mutated inputs. It means that faster fuzzers have more chances to discover vulnerabilities. Table 4 represents P^{2}IM is emulation at the register-level which makes fuzzing slow, so the number of executions is the lowest compared to the other two fuzzers. On the other hand, emulation in HAL-Fuzz is performed at the function-level, so faster fuzzing is possible, and as a result, the firmware is executed the most. At this time, CNC is not executed because there is no handler corresponding to the specific function (§VII). Finally, HEFF, a combination of P^{2}IM and HAL-Fuzz, shows approximately 0.9 times less execution result of P^{2}IM and approximately 2 to 4 times more execution than HAL-Fuzz. Moreover, Figure 3 shows that HEFF executed at least 2.7% and at most 9.6% over P^{2}IM.

This may be seen as a small speed increase, but each handler is at least 2 times and up to approximately 400 times faster than P^{2}IM when processing the same function (Table 5). In particular, handling HAL_Init at the function level can significantly reduce the time required. Nevertheless, because each handled HAL function is not



(a) CNC (b) PLC

(c) Robot (d) Drone

FIGURE 4. **Basic block coverage of HEFF and HAL-Fuzz.**

TABLE 5. **Average time for function handling in HEFF and P^{2}IM (μs).**

Function	HEFF	P^{2}IM
HAL_Init	5	2270
HAL_IncTick	5	1322
HAL_Uart_Init	11	4909
HAL_RCC_OscConfig	367	947
HAL_UART_Transmit	17	526
timers_init	12	1013
usart_init	290	504
serial_write	14	99

called more often than other functions when the firmware is executed, the fuzzing speedup is not significantly affected. However, this result means that using more handlers in HEFF, achieves acceleration (§VII). Furthermore, comparing otherwise, Table 7 shows that HEFF's firmware rewriting process reduces the model instantiation time up to 21 seconds at register level except for one (§VII). Therefore, HEFF is efficient because it can improve the speed even with a small number of handlers, and can also reduce the model instantiation time.

TABLE 6. **The number of basic blocks (5h) executed when fuzzing firmware 10 times.**

Firmware	HEFF	P^{2}IM		HAL-Fuzz	
	Median	Median	Improv.	Median	Improv.
CNC	1052	1067	-1.42%	254	+75.85%
PLC	707	519	+26.59%	252	+64.35%
Robot	1129	1166	-3.27%	315	+72.09%
Drone	1205	1278	-6.05%	199	+83.48%

C. RQ.2: SCALABILITY OF HEFF

To answer RQ2, we compared the basic block coverage with HAL-Fuzz and P^{2}IM. Table 6 shows the number of basic blocks of each fuzzer. Compared to HEFF, P^{2}IM shows a small difference from a minimum of 0.94 times to a maximum of 1.36 times, while HAL-Fuzz shows a big difference from a minimum of 3.58 times up to 6.05 times; during emulation, P^{2}IM shows the highest basic block coverage, and HAL-Fuzz shows the lowest basic block coverage. In addition, HEFF covers a large number of basic blocks compared to that in the case of HAL-Fuzz, despite handling approximately 10% of HAL functions for each firmware. Figure 4, which compares

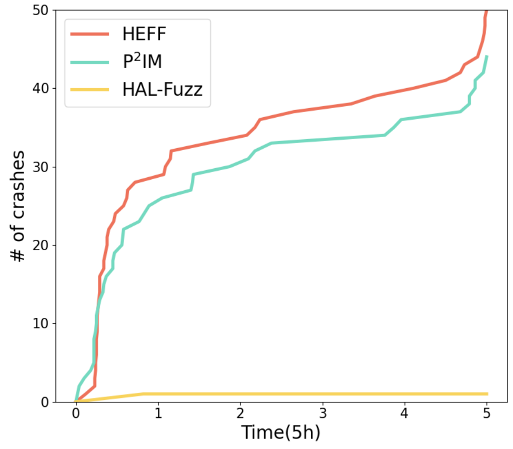


FIGURE 5. Number of crashes detected in 10 fuzzing runs (5h) of HEFF,P^2IM and HAL-Fuzz.

TABLE 7. Model instantiation time (median) required when performing fuzzing firmware 10 times.

Firmware	CNC	PLC	Robot	Drone
Dual-level modeling (s)	66	3	192	322
Register-level modeling (s)	65	6	201	343
Improve	+1.51%	-100%	-4.68%	-6.52%

HEFF's basic blocks with HAL-Fuzz, shows significant differences, thus, it means that HEFF is more scalable that HAL-Fuzz. In particular, in CNC, given that the "USART_putc" function, which is the transmit-related function, does not exist in the function list. Therefore, the subsequent operation is stuck and cannot be executed, and further fuzzer execution is impossible in HAL-Fuzz. By contrast, in HEFF, even if there is no function called in the function list for handler processing, because processing is possible at the register-level, it could operate firmware and perform fuzzing. Therefore, it was confirmed that a more scalable fuzzing was possible through HEFF.

D. RQ.3: EFFECTIVENESS OF HEFF

To validate the effectiveness of HEFF in finding bugs, we performed 10 fuzzings of each fuzzer on 4 real firmwares and compared them against P^2IM and HAL-Fuzz. As a result, HEFF, P^2IM, and HAL-Fuzz all detected bugs in PLC only among the 4 firmwares (CNC, PLC, Robot, Drone). Figure 5 shows the number of unique crashes resulting from 5(h) fuzzing on the PLC. These crashes are caused by memory corruption. The number of crashes was counted based on the last basic block of the test case that crashed during fuzzing, HAL-Fuzz found only one crash, while HEFF found a total of 49 crashes, which is six more than P^2IM. In addition, HEFF found the same number of crashes faster. Therefore, HEFF is effective because it can quickly detect more crashes.

VII. DISCUSSION

Remark that 19 out of 31 HAL function handlers (61%) are return-zero handlers [24]. This means that approximately 60% of the HAL functions do not affect the fuzzing performance. Accordingly, unlike HAL-Fuzz, HEFF does not implement function handlers that require complex logic but implements a return-zero handler for acceleration by connecting functions that do not affect firmware operation (Table 5). In addition, we implemented a receive handler for fuzzing. The HAL function handled by each experimental firmware is only approximately 10% of the total HAL functions called, and the frequency of the HAL functions called by each firmware is only approximately 0.24% 3% of the total function calls. However, despite the low frequency of calls, the fuzzing speed of the HEFF is improved (Table 4). That is, even a small effort of HEFF can affect the overall fuzzing speed. However, using many handlers, as implemented in HAL-Fuzz, faster speed improvement is expected. Moreover, Table 5 shows that Init functions require more time to process than other functions. Therefore, as more handlers are processed for Init-related functions, a greater speed improvement can be expected. Also, this improvement increases the possibility of finding more bugs.

Table 7 shows the model instantiation times of HEFF and P^2IM which does not use modified firmware. By running the modified firmware, HEFF can reduce the model instantiation time in contrast to P^2IM by not creating models for functions that are unnecessary for fuzzing. Therefore, the model instantiation times of the PLC, Robot, and Drone are faster than HEFF by 6.5% to 60% with respect to P^2IM. However, for CNC, HEFF is approximately 2% slower than P^2IM. This slight slowdown occurs because the HEFF's function address check logic makes it slower than the speed increased through the modified firmware. In other words, given that the function address check logic in HEFF monitors the changed PC value in real time, a tiny block overhead occurs. However, although the model instantiation time of HEFF is slower than P^2IM in CNC, its difference is the smallest in terms of model instantiation time difference with respect to other firmware. Moreover, despite this slowdown, the overall fuzzing speed increased (§VI-B).

VIII. CONCLUSION

To achieve fast and scalable firmware fuzzing, we proposed the new approach of dual-level peripheral modeling and implemented our prototype system called HEFF on top of P^2IM. We demonstrated its efficiency, scalability, and effectiveness by performing the experiments with P^2IM and HAL-Fuzz. In our prototype implementation, we adopted LibMatch and function handlers of HAL-Fuzz to deal with the abstract function level. Finally, we observed that HEFF was faster than P^2IM, more scalable than HAL-Fuzz, and found more bugs than P^2IM and HAL-Fuzz. Although HEFF was more efficient than P^2IM and more scalable than HAL-Fuzz in our experiments, there remain limitations. We expect the following future study for further improvement.

As we adopted LibMatch, we were able to handle only HAL functions at the abstract function level. The basic idea of HAL-Fuzz is splendid but limits scalability. We plan to develop a more general function matching mechanism to deal with more functions at the abstract function level and to relax the prerequisite of the exact environmental match, entailing even faster fuzzing than now. We also expect that this approach could address the mis-categorization problem at the register level; and in addition, it could handle the ''write'' functions, unnecessary for fuzzing, with a return-zero handler regardless of which register they access, for further improvement of fuzzing speed.

ACKNOWLEDGMENT
The authors thank Yeongpil Cho for fruitful discussion and feedback.

REFERENCES

[1] K. L. Lueth. (Nov. 2020). *State of the IoT 2020: 12 Billion IoT Connections, Surpassing non-IoT for the First Time*. Accessed: Jun. 1, 2021. [Online]. Available: https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/

[2] *National Vulnerability Database*. Accessed: Jun. 1, 2021. [Online]. Available: https://nvd.nist.gov/vuln/

[3] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *Proc. 13th Annu. Conf. Privacy, Secur. Trust (PST)*, Jul. 2015, pp. 145–152.

[4] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Jan. 2018, pp. 1–15.

[5] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *Proc. USENIX Secur. Symp.*, Aug. 2021.

[6] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *Proc. USENIX Secur. Symp.*, 2020, pp. 1201–1218.

[7] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proc. USENIX Secur. Symp.*, 2020, pp. 1237–1254.

[8] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 2123–2138.

[9] M. Zalewski. *American Funzz Lop*. Accessed: Jun. 1, 2021. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[10] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *Proc. 28th USENIX Secur. Symp.* Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 1099–1114.

[11] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, vol. 1, San Diego, CA, USA, Feb. 2016, p. 1.

[12] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: A case study on embedded web interfaces," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, May 2016, pp. 437–448.

[13] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, "FirmFuzz: Automated IoT firmware introspection and analysis," in *Proc. 2nd Int. ACM Workshop Secur. Privacy Internet Things (IoT S&P)*, Nov. 2019, pp. 15–21.

[14] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf., FREENIX Track*, Berkeley, CA, USA, Apr. 2005, p. 46.

[15] A. Beckus. (2012). *qEMU With an STM32, Microcontroller Implementation*. Accessed: Jun. 5, 2021. [Online]. Available: https://github.com/beckus/qemu_stm32

[16] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proc. 5th Program Protection Reverse Eng. Workshop*, Los Angeles, CA, USA, Dec. 2015, pp. 1–11.

[17] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, pp. 1–36, Apr. 2021.

[18] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2014, pp. 1–16.

[19] M. Kammerstetter, C. Platzer, and W. Kastner, "Prospect: Peripheral proxying supported embedded code testing," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur.*, Jun. 2014, pp. 329–340.

[20] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *Proc. USENIX Workshop Offensive Technol.*, Washington, DC, USA, Aug. 2015, pp. 1–10.

[21] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "AVATAR2: A multi-target orchestration platform," in *Proc. Workshop Binary Anal. Res.*, 2018, pp. 1–11.

[22] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna, "Toward the analysis of embedded firmware through automated re-hosting," in *Proc. Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, Beijing, China, Sep. 2019, pp. 135–150.

[23] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *Proc. USENIX Workshop Offensive Technol.*, Santa Clara, CA, USA, Aug. 2019, pp. 1–11.

[24] *HAL_Fuzz*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/ucsb-seclab/hal-fuzz

[25] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of DMA input channels for dynamic firmware analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 302–318.

[26] C. Spensky, A. Machiry, N. Redini, C. Unger, G. Foster, E. Blasband, H. Okhravi, C. Kruegel, and G. Vigna, "Conware: Automated modeling of hardware peripherals," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jun. 2021, pp. 95–109.

[27] (2019). *Embedded Markets Study*. Accessed: Jun. 1, 2021. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_20%19_Embedded_Markets_Study.pdf

[28] *STM32469I_EVAL Examples*. Accessed: Jun. 4, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I_EVAL/Examples

[29] *UART_HyperTerminal_IT Source Code*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I_EVAL/Examples/UART/UART_HyperTerminal_IT

[30] *UART_Printf Source Code*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I_EVAL/Examples/UART/UART_Printf

[31] *SPI_FullDuplex_AdvComPolling Source Code*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I-Discovery/Examples/SPI/SPI_FullDuplex_AdvComPolling

[32] *SPI_FullDuplex_AdvComIT Source Code*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I-Discovery/Examples/SPI/SPI_FullDuplex_AdvComIT

[33] *I2C_EEPROM Source Code*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I_EVAL/Examples/I2C/I2C_EEPROM

[34] *GPIO_IOToggle Source Code*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/STMicroelectronics/STM32CubeF4/tree/master/Projects/%STM32469I_EVAL/Examples/GPIO/GPIO_IOToggle

[35] *Radare 2*. Accessed: Jun. 6, 2021. [Online]. Available: https://github.com/radareorg/radare2

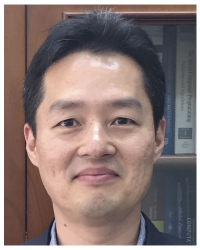
[36] *P^2IM Real-World Firmware Samples*. Accessed: Jun. 1, 2021. [Online]. Available: https://github.com/RiS3-Lab/p2im-real_firmware/tree/d4c7456574ce2c2ed03%8e6f14fea8e3142b3c1f7

EUNBI HWANG received the B.S. degree in statistics from Sungshin Women's University, Seoul, South Korea, in 2019. She is currently pursuing the Ph.D. degree with the Information Security Laboratory, Yonsei University, Seoul. Her research interests include software security, system security, and the IoT security.

HYUNSEOK LEE received the B.S. degree in computer engineering from Yonsei University, Wonju, South Korea, in 2020. He is currently pursuing the M.S. degree with the Information Security Laboratory, Yonsei University, Seoul. His research interests include mobile, the IoT security, digital forensic, and app developing.

SEYEON JEONG received the B.S. degree in information security from Daegu Catholic University, Gyeongsangbukdo, South Korea, in 2018. He is currently pursuing the M.S. degree with the Information Security Laboratory, Yonsei University, Seoul. His current research interests include software security, system security, and the IoT security.

MINGI CHO received the B.S. degree in computer engineering from Pusan National University, Busan, South Korea, in 2017. He is currently pursuing the Ph.D. degree with the Information Security Laboratory, Yonsei University, Seoul. His research interests include software and system security, fuzzing, and memory safety.



TAEKYOUNG KWON (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from Yonsei University, Seoul, South Korea, in 1992, 1995, and 1999, respectively.

From 1999 to 2000, he was a Postdoctoral Research Fellow with the University of California, Berkeley, CA, USA. From 2001 to 2013, he was a Professor of computer engineering with Sejong University, Seoul. He is currently a Professor of information security with Yonsei University, where he is also the Director of the Information Security Laboratory. His research interests include authentication, cryptographic protocols, network security, software and system security, usable security, and adversarial machine learning.

Dr. Kwon is a member of ACM and USENIX. He serves on the Director Board Member for Korea Institute of Information Security and Cryptology. He serves on the Editorial Committee Member for the Korean Institute of Information Scientists and Engineers.