FIRM-COV: High-Coverage Greybox Fuzzing for IoT Firmware via Optimized Process Emulation

JUHWAN KIM¹, JIHYEON YU¹, HYUNWOOK KIM², FAYOZBEK RUSTAMOV², AND JOOBEOM YUN¹

¹Department of Computer and Information Security, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul 05006, South Korea
²Department of Computer and Information Security, Sejong University, Seoul 05006, South Korea

Corresponding author: Joobeom Yun (jbyun@sejong.ac.kr)

This work was supported by the Ministry of Science and ICT (MSIT), South Korea, through the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2021-2018-0-01423 and Grant IITP-2021-2020-0-01602.

ABSTRACT With the growing prevalence of the Internet of Things (IoT), related security threats have kept pace. The need to dynamically detect vulnerabilities in IoT devices cannot be overstated. In this work, we present FIRM-COV, the first high coverage-oriented greybox fuzzer for IoT firmware. FIRM-COV leverages newly optimized process emulation by targeting IoT programs and mining real-world vulnerabilities. FIRM-COV focuses on solving problems of IoT fuzzing based on empirical analyses, using the required structured input, the inaccuracy and instability of emulation, and the required high code coverage. By optimizing the existing emulation technique, FIRM-COV always maintains a stable state and achieves high accuracy when detecting vulnerabilities. We also implement a dictionary generation algorithm to provide structured input values and synergy scheduling to achieve high coverage and throughput. We compare FIRM-COV with other IoT fuzzing frameworks for eight real-world IoT devices. As a result, FIRM-COV achieves the highest coverage and throughput, finding the fastest and most 1-day vulnerabilities with almost no false-positives. It also found two 0-day vulnerabilities in real-world IoT devices within 24 h.

INDEX TERMS Dictionary, emulation, firmware, fuzzing, IoT, vulnerability.

I. INTRODUCTION

With the growing prevalence of Internet of Things (IoT) devices, security threats are increasing simultaneously. According to a Cisco report [1], there will be about 28.5B internet-connected devices worldwide by 2022, with more than half expected to provide IoT services. Obviously, IoT devices can be easily targeted by remote attackers. For example, an anonymous Russian hacker compromised nearly 73,000 internet-protocol (IP) cameras worldwide, exposing them on the Insecam website. This highlights the extant risk to public facility and control systems and medical devices, from which a mass-casualty event could be caused.

Related studies [2]–[10] have proposed approaches to identifying IoT-device vulnerabilities caused by weak authentication, insecure management, and memory corruption. In particular, general-purpose IoT devices contain freely accessible firmware that controls hardware and systems. Such firmware can be exploited. Thus, those vulnerabilities need to be assessed, located, and patched before the remote attackers can take full advantage.

Fuzzing is an efficient and powerful method of detecting software vulnerabilities [11], and it does so while automatically generating input values to the firmware while monitoring it for abnormal behavior. Fuzzing does not limit the range of vulnerabilities that can be detected. However, depending on time and performance, it can find unexpected ones. Therefore, in this work, we design and implement an emulation-based IoT fuzzing system that detects potential memory corruptions in IoT devices. It emulates IoT firmware without requiring real-world devices. However, it targets IoT programs and mines real vulnerabilities.

IoT fuzzing is challenging for the following reasons. First, some IoT programs can only accept structured input values. In such a case, the server-side can discard unstructured input values. Second, stable and accurate fuzzing cannot be performed for some IoT programs. Because emulation-based approaches do not fully account for all hardware, reading and writing data from peripheral devices (e.g., non-volatile random-access memory (NVRAM)) can cause the device
to crash. Furthermore, input values lacking responses from the server (e.g., memory corruptions) may be classified as crashing input values. Finally, IoT fuzzing must achieve efficient code coverage. Unfortunately, unlike desktop fuzzing approaches, existing IoT approaches normally cannot do this. Thus, it must redundantly explore the same paths or a set of mutation operators that most often do not provide satisfactory results.

Our solution provides a high-coverage greybox fuzzing technique for IoT firmware via optimized process emulation. In this paper, we achieve four design goals simultaneously. For availability, our system can operate without real-world IoT devices. For accuracy, our system can find vulnerabilities with almost no false-positives. For stability, our system can detect system’s panic state. For efficiency, our system achieves high coverage and throughput.

The core techniques leverage a dictionary generation algorithm, optimized process emulation, and synergy scheduling. Our system generates structured input values receivable by IoT programs and improves the scheduling of fuzzing to achieve high coverage and throughput while detecting the system’s panic state to maintain emulator stability. We implement our firmware-coverage (FIRM-COV) framework based on the techniques mentioned above. From a users’ perspective, FIRM-COV performs effective fuzzing for IoT devices having only Linux-based firmware.

We compare FIRM-COV with other IoT fuzzing frameworks, including a benchmark program and eight real-world IoT devices. The dictionary generation algorithm has negligible performance overhead and is effective for many IoT programs. The optimized process emulation improves stability and accuracy. The scheduling technique achieves high code coverage and fuzzing throughput. FIRM-COV finds the most 1-day vulnerabilities and does so faster than other frameworks. It notably found two 0-day vulnerabilities in a real-world IoT test.

Contributions. We summarize the contributions of the paper as follows:

- We show that some IoT programs can only receive structured input values. We further investigate the root cause of this phenomenon.
- We propose a new dictionary generation algorithm that performs static analysis for efficient input generation.
- We propose a new synergy scheduling technique that achieves high coverage and throughput.
- We design and implement FIRM-COV, an IoT fuzzing framework based on optimized process emulation, and we apply high coverage-oriented fuzzing to IoT firmware for the first time.

Roadmap. The rest of this paper is organized as follows. Section II gives the background of firmware emulation and our insights of FIRM-COV with a running example. Section III and IV present the detailed design of FIRM-COV. The evaluation results are summarized in Section V. Section VI discusses some limitations of the current design. Section VII reviews the related work, and finally Section VIII concludes this paper.

II. BACKGROUND AND MOTIVATION

A. TYPES OF FUZZING

Fuzzing [11] is an automated testing technology that discovers software bugs and vulnerabilities by sending randomized input values to the target program while monitoring its reactions. Furthermore, it is faster and more accurate than manually reviewing the source code in most cases. Such fuzzing have been widely adopted by bug hunters and software developers because of their simplicity and scalability [12]. However, general fuzzing has difficulty finding deeply buried bugs. Thus, deep fuzzing requires greater efficiency. Such fuzzers are usually classified into grammar- [13]–[17] and mutation-based [18]–[29] approaches, depending on test-case generation.

1) Grammar-based fuzzing. This applies the design, format, and protocol of the target. Test cases are generated by a defined input grammar, and the probed areas are specified in advance. This fuzzing method can cover all pre-defined input parts, but it has the disadvantage that the preparatory work is difficult and time consuming. Therefore, for large-scale programs, the input grammar may be too complicated.

2) Mutation-based fuzzing. This method generates test cases based on past cases. During the first iteration, the input value (i.e., seed) is mutated, and from subsequent iterations, it changes based on the generated prior test cases. Thus, the mutation-based fuzzing method does not require the initial preparatory work as grammar-based fuzzing, because it learns the format of the target itself. However, general mutation-based fuzzers can take longer to learn the target’s format and are only partially useful if the target receives structured input values.

From another perspective, fuzzing approaches are divided by their methods of execution monitoring. Traditional fuzzing employs a blackbox approach, which does not explore inside the target program. Subsequently, improved execution monitoring methods have been proposed, including whitebox fuzzing [23], [27], [29], [30], which performs fuzzing while solving constraints inside the target program. Greybox fuzzing [19]–[22], [24]–[26], [28] performs fuzzing while collecting only some information about the target program.

Our present research leverages mutation-based fuzzing to detect vulnerabilities in IoT devices, and we use the greybox approach to achieve high code coverage.

B. OBSTACLES IN FIRMWARE EMULATION

As shown in Fig. 1, general-purpose IoT devices contain firmware that interacts with the hardware. Hence, remote attackers can easily exploit firmware vulnerabilities. Custom vendor-developed applications included in the firmware are
not open source and are rarely publicly evaluated. Thus, they are prone to vulnerabilities [8]. Therefore, we adopt an emulation-based environment. We first discuss existing emulation techniques implemented using QEMU, a generic and open-source machine emulator and virtualizer [31].

1) **User-mode emulation.** This emulation runs a program compiled using a different instruction set than the host machine, which is aimed at fast cross-compilation and debugging. It first finds the main function of the target program and executes. AFL-QEMU [20] fuzzing adopts this technique, and it performs fuzzing while measuring code coverage via emulation, even without the source code of the target. However, user-mode emulation does not recreate hardware devices that can crash. In other words, it is faster than system-mode emulation, but it is not a perfect abstraction.

2) **System-mode emulation.** This emulation is a virtual machine that provides a diverse set of hardware (e.g., peripherals) and emulates an entire system. It first finds the first instruction to be executed by the central processing unit (CPU) after it is reset. It then works from this point. Applications based on this mode include FIRMADYNE [4], DECAF [32], and PANDA [33]. FIRMADYNE is a full-system automation emulation that performs vulnerability checks by combining dynamic analyses. DECAF and PANDA are platforms that combine this mode with dynamic analysis to analyze specific binaries during emulation. Thus, system-mode emulation can be applied, owing to its superior compatibility. However, system-mode emulation has a considerable performance overhead, because all hardware is implemented in software. Hence, it requires constant optimization.

3) **Unicorn engine.** The unicorn engine [34] is a QEMU-based CPU emulator that removes other devices and emulates only the CPU via software. It also executes different instruction sets than the host machine. It is more independent, flexible, and concise, and it has a lighter load than QEMU. It thus provides an interface for dynamic instrumentation. FIRM-CORN [10] uses the unicorn engine to perform optimized virtual execution of the target. However, this engine must dump context information from real-world IoT devices while resorting to the CPU context. Thus, the unicorn engine always requires physical devices.

4) **Augmented process emulation.** This was proposed by FIRM-AFL [9] and aims for high efficiency and compatibility by combining user- and system-mode emulation. It mainly executes the target as an emulated user on the host machine. When system calls cannot be handled, it switches to system-mode emulation and processes the calls. However, the augmented process emulation does not recover if the system falls into an infinite loop or reboots because of repeated fuzzing input values. Therefore, it needs stable emulation.

In summary, not all existing emulation techniques are suitable for emulating general-purpose IoT firmware. Thus, we must implement a novel, efficient, and stable method. Therefore, in this paper, we adopt an augmented process emulation to design an emulation-based IoT fuzzing technique and improve it so that we obtain a stable and efficient emulation method.

**FIGURE 2. Structure of HTTP request header.**

**C. CHALLENGES OF IoT FUZZING**

We have thus far identified the major challenges of IoT fuzzing based on the methodologies of extant studies and our empirical analysis of general-purpose IoT devices. Because these challenges make IoT fuzzing difficult and less efficient, we are required to solve the following challenges.

- **[C1] Structured input values.** The first challenge of IoT fuzzing relates to network-based IoT devices only receiving structured input values, as shown in Fig. 2. The HTTP request header comprises a start-line, various headers, and a body. The start-line contains the method, the request-target, and the hypertext transfer protocol (HTTP) version. Generally, these methods represent GET or POST, and the request-target represents...
a file on the server-side. The HTTP header structure is a set of keys and values, and the body is used for
the POST or PUT methods to send parameter values
to the server. In the case of other methods, such as GET,
the body is not required.
As a result, we must structure the inputs well to fuzzify
these server-side applications. Otherwise, they will be
discarded. As shown in Listing 1, most common gate
interface (CGI) programs do not take standard input,
nor do they retrieve data sent by the client through the
getenv() function. If the method’s value is missing
or invalid in the data, the data is discarded from the
server. Thus, the client cannot explore deep paths.

- [C2] Panic state. The second challenge of IoT fuzzing is the accuracy and stability of the emulator. The
emulation-based approach implements all possible hard-
ware via software, but it is not a complete implementa-
tion. Thus, if it reads and writes from a peripheral device
(e.g., NVRAM), it may cause a crash. Furthermore,
它可以 classify input values that do not respond from the
server. In the case of other methods, such as GET,
the POST or PUT methods to send parameter values
to the server. Therefore, we must structure the inputs well to fuzzify
these server-side applications. Otherwise, they will be
discarded. As shown in Listing 1, most common gate
interface (CGI) programs do not take standard input,
nor do they retrieve data sent by the client through the
getenv() function. If the method’s value is missing
or invalid in the data, the data is discarded from the
server. Thus, the client cannot explore deep paths.

- [C3] Code coverage. The third challenge of IoT fuzzing is code coverage used for efficiency. Existing
research [35] has shown that a 1 % increase in code
coverage can increase bug discovery by 0.92 %. Unfortu-
nately, apart from desktop fuzzing approaches, con-
ventional IoT fuzzing approaches have no studies that
have achieved high code coverage. Thus, they have not
explored most of the same paths, nor have they chosen
a mutation operator that is inefficient at finding crashes
and paths.

Solutions. We propose the following solutions to the chal-
lenges above.

- Mutating inputs with dictionary. We use a dictionary-
based AFL mutation [20] to generate structured input
values receivable by IoT programs. It selectively allows

the user to improve the target program syntax-blind and
utilizes the user’s syntax tokens as input generation. The
AFL automatically identifies these syntax tokens so that
structured input values can be generated.

- Detecting panic state via exception processing. We implement an exception processing module inside
the emulator to detect the device’s panic state in
real time. During fuzzing, it always returns hardware-
dependent functions as true, and inputs that are not
responded from the server are discarded from the input
queue via a timeout. Additionally, it detects the reboot
of the emulator and restores it to the initial state of the
system. Thus, we can always keep the emulator.

- Improving fuzzing scheduling. We use the greybox
fuzzing approach and improve the scheduling process
to achieve high code coverage. Moreover, we improve
and combine both seed and mutation scheduling of the
fuzzer, as inspired by existing studies [25], [28]. They
have already demonstrated effective desktop fuzzing.
Thus, we expect to achieve high code coverage for IoT
fuzzing.

III. PRE-ANALYSIS IoT Firmware
In this section, we describe the pre-analysis IoT firmware,
the first layer of FIRM-COV. This step configures the emu-
lation environment for the IoT firmware and performs static
analysis for effective input generation. We also show the
importance of dictionary-based mutation in IoT fuzzing.

A. Configuring the Environment for Emulation
We adopt an emulation-based approach to detect vulnerabil-
ities in real-world IoT devices. It can detect them without
physical access and can perform stable and accurate analysis
with only general-purpose IoT firmware.

Several firmware types are available in the IoT. Thus,
we acquire them and configure an emulation environment
based on the FIRMADYNE [4] framework: an automated
system for performing emulation and dynamic analysis of
Linux-based firmware based on the QEMU [31] emulator.
We switch the existing QEMU with the state-of-the-art ver-
version [32] for better scalability and efficiency, and we use a
customized kernel for stable firmware execution. The state-
of-the-art QEMU can monitor a specific running process
while implementing and calling custom callback functions.
More details are provided in Section IV.

More specifically, our system first extracts the compressed
file system from the Linux-based firmware image using
Binwalk [36] and collects architecture information with the
busybox application. Then, the extracted file system is cre-
ated as a QEMU image and is emulated. Then, a customized
kernel (e.g., MIPSEL, MIPSEB, or ARM) is emulated, and
a virtual network interface for network communication is
created.

B. Dictionary Generation Algorithm
As mentioned, some IoT programs can only receive struc-
tured input values. If we do not consider structuring during

Listing 1. Example code of Common Gate Interface (CGI) program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

int main(void)
{
    char *var;
    char buf[256];
    if (getenv("REQUEST_METHOD") == NULL ||
        strcmp("GET", getenv("REQUEST_METHOD")) ||
        strcmp("POST", getenv("REQUEST_METHOD")))
    {
        printf("Failed check.\n")
        return 1;
    }
    ... return 0;
}
```

J. Kim et al.: FIRM-COV: High-Coverage Greybox Fuzzing for IoT Firmware via Optimized Process Emulation
the input generation of IoT fuzzing, the server-side may discard the input values. To demonstrate this, we compared two systems with and without the dictionary-based mutation applied to the extant study of FIRM-AFL [9] as listed in Table 1. The evaluation environment is as follows. We fuzzed for 30 min for each program—the initial seed was an unstructured meaningless value (e.g., fuzz), and the dictionary file used for dictionary-based mutation was provided by the author of FIRM-AFL.

The two systems used the same fuzzer tool, but the total paths differed significantly, owing to differences in the internal algorithm. Notably, the FIRM-AFL without dictionary-based mutation found only one discarded path from the server among all programs on the DIR-815 device. However, FIRM-AFL with the dictionary-based mutation found many more, even when given a meaningless seed. Additionally, FIRM-AFL without dictionary-based mutation found more paths in the DIR-825 device compared with others. However, these paths did not include the hidden headers defined in the server and were all unstructured input values. As a result, we still need to use a dictionary-based mutation for overall efficiency.

**IV. EMULATION-BASED IoT GREYBOX FUZZING**

In this section, we describe the second step of FIRM-COV: emulation-based IoT greybox fuzzing. This step is divided
A. OPTIMIZED PROCESS EMULATION

We propose an optimized emulation of IoT firmware to detect vulnerabilities without requiring real-world devices by applying two emulations, as shown in Fig. 4 and inspired by the existing augmented process emulation [9]. It generally executes the target program in user-mode emulation for efficiency. Only when exceptions are caused in the system is it switched to full-system emulation to handle exceptions. The features of the optimized emulation are as follows:

1) **Generate entry state for IoT program.** Most fuzzers execute the target at a specific point (e.g., the entry point of `main()` function) for efficient fuzzing. Then, it creates a child process of the target using a `fork()` function, and it fuzzes this child process. This approach skips the process’s initial environmental setting and is made effective by feeding the input value to the target repeatedly. Therefore, we set the target program’s entry point that receives the network packet from the client as a specific point, and we abstractly store the state of the target program executed up to that point. The state contains memory, registers, and various other information, including open files. This gives us our entry state.

We first initialize the entry state using full-system emulation. However, it is the virtual machine that emulates the entire system, not just a specific program. Therefore, we capture the target program using state-of-the-art QEMU [32] capable of dynamic introspection. More specifically, we create our own callback function inside the full-system emulation. First, the client-side sends a request packet to the target program in the server through the server-side virtual network interface. Second, the target program is called.
because it must receive the request packet. Finally, its own callback function captures the state of the target program upon receipt. As a result, the entry state generated in full-system emulation is synchronized with user-mode emulation. Then, the system is ready for fuzzing.

2) **Snapshot for scalability of emulation.** As mentioned, most fuzzers make copies of the target program and fuzz those for efficiency. We adopt this approach. Thus, we must create copies of both emulators. User-mode emulation is easy to multi-process and copy. Nevertheless, a full-system emulation is intuitively a virtual machine. Hence, it is very expensive to create copies. Furthermore, rebooting a full-system emulation requires a significant amount of time [5]. Therefore, our user-mode emulation uses the `fork()` function to create copies of the target program while full-system emulation uses its snapshot for scalability. Thus, full-system emulation loads the entry state through the snapshot every time an input value of fuzzing is processed.

3) **Eliminate panic state.** The emulation-based approach interacts with fuzzing. Thus, it should always remain stable. Our emulation is based on existing approaches [9] that can flexibly manage page faults. However, it might still invoke the panic state of the emulator. For example, user-mode emulation causes a page fault if the accessed memory address does not exist when executing the target program. Our system detects the page fault and requests a new memory mapping from the full-system emulation. However, the emulation itself may cause a user process to access the kernel address space when processing a faulting instruction. This may cause a reboot, or the user process may be terminated. In any case, the two emulations fall out of sync, and the system fails. Furthermore, the signal handler included in the emulator may handle input values that do not respond to vulnerabilities, such as memory corruptions. Therefore, we inject our exception processing module into the emulator to detect the panic state that cannot respond to fuzzing. More specifically, it checks whether the target is executed in user-mode emulation using a periodic liveness check. When the panic state is detected, we re-synchronize the two emulators, and the input value that triggered the panic state is discarded.

4) **Handle hardware-dependent functions.** One of the major challenges of IoT fuzzing is handling hardware-dependent functions. This refers to functions that access peripheral devices (e.g., NVRAM) included in IoT devices. For example, some firmware types use a method of fetching and storing information from config files found in a peripheral device when executing a process. Fig. 5 shows that when the firmware boots up, the `nvram_get()` function is called to retrieve information from the NVRAM device to configure the initial environment. If the emulator does not handle these hardware-dependent functions, it will crash. Therefore, we implemented a library file by manually analyzing all the firmware types evaluated to handle these hardware-dependent functions. Then, the emulator loads it with the environment variable, `LD_PRELOAD`. As a result, we can see that the library’s memory area that can hijack hardware-dependent functions is properly mapped when our system creates the entry state for a specific IoT program, as shown in Fig. 6.

**B. COVERAGE-ORIENTED FUZZING**

We propose coverage-oriented fuzzing for efficiency. It interacts with the emulation and can reach deep paths by collecting code coverages with a greybox fuzzing technique. Furthermore, it uses a dictionary-based mutation to generate structured input values and introduces synergy scheduling to assign priorities and mutation operators. The features of coverage-oriented fuzzing are as follows:

1) **Code coverage collection.** Traditional blackbox fuzzing achieves high fuzzing throughput, but it usually cannot explore deep paths, because it does not analyze the inside of the target program. Therefore, we adopt the...
greybox fuzzing approach that performs fuzzing while collecting code coverage. Thus, we determine whether mutated input values have reached new paths and store the information about the collected code coverage in the bitmap implemented by the AFL [20]. As a result, we can instrument the code coverage through a bitmap without significant overhead (e.g., static analysis). The code coverage gradually increases as only input values reaching new paths are added to the queue directory of the fuzzer.

2) **Dictionary-based mutation.** Most mutation-based fuzzers do not account for the grammar of the target. Hence, they suffer from targets (e.g., HTTP or SQL) that can only receive structured input values. For example, it is not easy to change the simple input value “fuzz” to “GET/HTTP 1.1” through mutation. In other words, exploring targets that can only receive simple structured input values requires a great deal of time and is limited to discovering deep bugs. Therefore, we adopt the dictionary-based mutation approach implemented by AFL to generate structured input values, which is efficient for real-world IoT programs (Section III-B), and it reads the dictionary generated by the proposed dictionary generation algorithm to generate structured input values.

3) **Synergy scheduling.** Recent fuzzing studies have shown that traditional desktop fuzzing approaches [25], [28], [38] schedule various fuzzing factors for efficiency, whereas IoT fuzzing approaches are not conducted with scheduling. Most IoT fuzzers execute the same paths and mutate test cases with inefficient mutation operators. We improve the seed and mutation scheduling and add synergy scheduling that combines them, as shown in Fig. 7. We provide an algorithm that prioritizes generated test cases and mutation operators for efficient and high code coverage. Further, we implement the fuzzer module based on AFL [20]. The details of the improved scheduling are as follows:

- **Seed scheduling.** AFL’s power scheduler generates too many inputs from test cases executing high-frequency paths. Thus, it is inefficient and takes too long. It calculates the performance score of input \( i \) as follows:

\[
p(i) = \alpha(i)
\]

Therefore, we apply a low-frequency based algorithm [25] for seed scheduling and technically modify the calculation of the power scheduler. The improved AFL power scheduler calculates the performance score of input \( i \) as follows:

\[
p(i) = \min\left(\frac{\alpha(i)}{\beta}, \frac{2s(i)}{f(i)} \right)
\]

where \( s(i) \) refers to the number of times input \( i \) is selected from the queue of the fuzzer. \( f(i) \) refers to the number of input values executed in the same paths as input \( i \). As a result, we can identify test cases with low path reachability according to the performance score and generate sufficient input values from them.

- **Mutation scheduling.** AFL’s mutation scheduling consists of three stages (i.e., deterministic, havoc, and splicing). There are 11 mutation operators as shown in Fig. 8. However, those operators have different efficiencies when detecting paths and crashes, depending on the target program. Thus, an operator that performs well in the current test case may not perform well in the next one. Therefore, we apply a custom particle swarm optimization-based algorithm [28] for mutation scheduling. It can gradually select efficient mutation operators by finding the optimal probability distribution through an iterative loop. A previous study [28] showed that AFL was inefficient, because it consumed too much time during the deterministic stage. Therefore, we accelerate the speed of fuzzing by performing sufficient dictionary-based mutation during the deterministic stage and skipping it for all test cases if no new paths or crashes are found for 5 min. Hence, we can select more test cases from the fuzzing queue and discover potential vulnerabilities faster than traditional AFL.

V. IMPLEMENTATION AND EVALUATION

In this section, we propose the implementation framework for FIRM-COV and analyze the evaluation results...
 TABLE 3. Program information used for evaluation.

| ID | Model | Vendor | Version | Device | Program | Size (KB) | CPU Arch | Description |
|----|-------|--------|---------|--------|---------|----------|----------|-------------|
| 1  | DIR-825 | D-Link | 2.02NA  | Router | httpd   | 444.8    | MIPSBE   | Embedded HTTP Server |
| 2  | DIR-850L | D-Link | 1.03    | Router | hnap    | 173.9    | MIPSBE   | CGI Binary Program |
| 3  | DAP-2695 | D-Link | 1.11RC044 | Router | httpd   | 166.2    | MIPSBE   | Embedded HTTP Server |
| 4  | DIR-815 | D-Link | 1.01    | Router | kedwig.cgi | 129.4 | MIPSBE   | CGI Binary Program |
| 5  | DIR-871LW | D-Link | 1.00B05 | Router | hnap    | 177.9    | MIPSBE   | CGI Binary Program |
| 6  | TEW-632BRP | Trendnet | 1.10B32 | Router | httpd   | 321      | MIPSBE   | Embedded HTTP Server |
| 7  | TV-IP110WN | Trendnet | 1.2.68 | Camera | video.cgi | 9.2 | MIPSBE | CGI Binary Program |
| 8  | WR940N | TP-Link | V4      | Router | httpd   | 1700     | MIPSBE   | Embedded HTTP Server |

TABLE 4. nbench [42] results of both emulations. Test unit is iterations/second.

| Benchmark         | Optimized process emulation | Augmented process emulation [9] | Increase |
|-------------------|------------------------------|---------------------------------|----------|
| Numeric sort      | 781.8                        | 777.8                           | 0.51%    |
| String sort       | 90.631                       | 90.522                          | 0.12%    |
| Bitfield          | 2.7484e+08                   | 2.7442e+08                      | 0.15%    |
| FP emulation      | 259.8                        | 259.76                          | 0.01%    |
| Fourier           | 2043.7                       | 2063.3                          | -0.94%   |
| Assignment        | 18.985                       | 19.024                          | -0.2%    |
| IDEA              | 4499.6                       | 4499.5                          | -0.02%   |
| Huffman           | 1554.7                       | 1457.8                          | 6.64%    |
| Neural Net        | 2.8392                       | 2.8375                          | 0.06%    |
| LU Decomp         | 76.702                       | 76.667                          | 0.04%    |

in detail. Section V-A provides detailed information about implementation. Section V-B introduces the evaluation setup (i.e., dataset and experimental environment) and discusses our evaluation questions. From Section V-C onwards, evaluation questions are answered based on the evaluation results.

A. FRAMEWORK IMPLEMENTATION
We have implemented FIRM-COV using python and C language for various libraries and hardware access. We also integrated several open-source projects.

1) Pre-analysis IoT firmware phase. Prior to fuzzing, we implemented a full-system emulator for a specific firmware using FIRMADYNE [4] and DECAF [32]. This makes dynamic analysis of IoT programs easier by replacing QEMU [31], which is used in FIRMADYNE, with DECAF. Additionally, after loading the target IoT program into IDA Pro [39], we implemented an automation script that extracts only interesting tokens and generates a dictionary file for fuzzing using the IDAPython module [40].

2) Emulation-based IoT greybox fuzzing phase. During this stage, we implemented the optimized process emulator by integrating the user-mode QEMU with the full-system emulator mentioned above. It measures code coverage for the target program via instrumentation and includes functions that can handle multiple exceptions. We also implemented an AFL fuzzer [20] that can discover vulnerabilities in target programs.

B. EVALUATION SETUP
1) IoT devices selection. We selected eight IoT devices for evaluation using the FIRMADYNE dataset [41] and acquired the IoT firmware of these devices from their official websites. The device types included routers or IP cameras and were grouped under three vendors. Because they are network-related IoT devices, most provide user functions through the network. From the perspective of remote attackers, network-related programs are likely to be targets. Therefore, we used eight network-related IoT programs for evaluation, as shown in Table 3.

2) Experimental environment. We constructed various IoT fuzzing frameworks based on the open-source FIRM-AFL [9] to demonstrate our system’s excellence. We used them for comparative evaluation, and the core functions of each framework are as follows.

(a) Baseline: We used the existing research FIRM-AFL as the baseline, which comprises AFL [20] and augmented process emulation.
(b) FIRM-AFLFast: In this configuration, we switched the existing power schedule of AFL to AFLFast [25] to guide fuzzing to low-frequency paths.
(c) FIRM-MOptAFL: In this configuration, we switched the existing mutation scheduling of the AFL to the mutation scheduling proposed by MOpt-AFL [28] to select efficient mutation operators. It also skips the deterministic stage (i.e., the first stage of mutation scheduling) if paths and crashes are not found for a given threshold time (i.e., 5 min).
(d) FIRM-COV: This is our proposed system. In this configuration, we changed the existing emulation approach to optimized process emulation. We introduced synergy scheduling and applied it to the AFL.

AFL can easily search the grammar of the target program using the user-generated dictionary file. We then generated each dictionary file for all target IoT programs using our proposed dictionary generation algorithm and provided the corresponding dictionary file to all IoT fuzzing frameworks. Similarly, we generated each initial seed for all target IoT programs, and all performed evaluations using the same seed. We performed 24-h fuzzing for all target IoT programs, and all evaluation results were the average of three measurements. We used a desktop environment with an Intel i5 processor with 32-GB of RAM, and the operating system was Ubuntu 16.04 LTS. The version of both state-of-the-art...
TABLE 5. Dictionary generation result.

| ID | Model     | Vendor | Program  | Strings (Num) | Tokens (Num) | Portion | Overhead (Second) |
|----|-----------|--------|----------|---------------|--------------|---------|------------------|
| 1  | DIR-825L  | D-Link | httpd    | 2682          | 457          | 17.03%  | 60.93            |
| 2  | DIR-855L  | D-Link | hnap     | 533           | 118          | 22.13%  | 10.77            |
| 3  | DAP-369S  | D-Link | httpd    | 734           | 100          | 13.62%  | 9.59             |
| 4  | DIR-815   | D-Link | hedwig.cgi| 317           | 81           | 25.55%  | 5.17             |
| 5  | DIR-817LW | D-Link | hnap     | 534           | 118          | 22.09%  | 8.77             |
| 6  | TBS-632BRP| Trendnet| httpd    | 2207          | 399          | 18.07%  | 22.64            |
| 7  | TV-IP110WN| Trendnet| video.cgi| 26            | 12           | 46.15%  | 1.53             |
| 8  | WR940N    | TP-Link| httpd    | 8874          | 422          | 4.75%   | 192.59           |

QEMU [32] and user-mode QEMU was 2.1.0, and the version of AFL was 2.52b.

3) Evaluation questions. We answered four evaluation questions:
   - EQ1: What is the performance overhead of FIRM-COV?
   - EQ2: How effective are our approaches?
   - EQ3: What is the code coverage of FIRM-COV?
   - EQ4: What is the vulnerability detection capability of FIRM-COV?

C. PERFORMANCE OVERHEAD (EQ1)

1) CPU benchmark. The emulator implements all hardware, including peripheral devices in the software. Thus, measuring the emulator’s performance overhead was a significant operation. We performed testing with the nbench benchmark program [42] to compare the performance of our proposed emulation technique to previous ones [9]. nbench is a CPU benchmark that provides indices for integers, floats, and memory performance. Because the emulation techniques used for evaluation use two emulators (i.e., full and user), they must synchronize the target program’s entry state. Therefore, we modified the code to generate and synchronize the entry state from nbench’s main function. Then, each emulation technique was used to execute nbench and measure the results.

The evaluation results are shown in Table 4. We can see that our proposed emulation technique has little performance overhead compared with the existing emulation technique. FIRM-AFL [9] demonstrated that augmented process emulation does not significantly differ in performance from user-mode emulation. Therefore, we can see that the optimized process emulation is also efficient. However, because nbench is a relatively simple benchmark program and does not cause many exceptions, we will need to evaluate the system using several benchmark programs in the future.

2) Dictionary generation algorithm. As discussed, just using the dictionary-based mutation shown in Table 1 is quite useful for IoT fuzzing. Thus, we applied an approach to extracting the target IoT program’s internal strings and configuring them into a dictionary. However, configuring unrelated code coverage is inefficient, because it incurs overhead on mutation scheduling. Therefore, we tokenize only strings whose code coverage can be increased through our proposed dictionary generation algorithm. Then, we measure its performance overhead.

The evaluation results are shown in Table 5. Most IoT programs tokenized only a minimum number of strings. For example, in the WR940N model, only 4.75% of the strings were tokenized from all extracted. Additionally, the time required for the proposed algorithm to configure the dictionary was proportional to the total number of extracted strings. For example, the TV-IP110WN model having the least number of extracted strings took 1.53 s. The WR940N model having the highest number of extracted strings took 192.59 s, and the average of all models used for evaluation took 37.8 s. This performance overhead is acceptable, given the discovery of vulnerabilities and improved code coverage allowed by the dictionary generation algorithm.

D. EFFECTIVENESS OF OUR APPROACHES (EQ2)

1) Availability and stability. To evaluate the availability and stability of the optimized process, we selected 150 network-related IoT devices based on the Linux kernel using the FIRMADYNE dataset [41]. We acquired the corresponding firmware images from official websites. Then, we automated everything—from firmware extraction to emulation—to ensure that the firmware would emulate normally. About 30 firmware accessed the NVRAM device during the initial boot process, but it was not a problem, because we loaded the library to provide the NVRAM-related functions. As a result, we confirmed that all emulated firmware booted normally and that several services were working. Our system built an environment without real-world IoT devices and performed stable firmware emulation without crashing.

2) Fuzzing throughput. A fuzzing-based system can explore deeper paths and discover various vulnerabilities when using dynamic analysis techniques. However, the fuzzing throughput of a system cannot be high without considering optimization, owing to overhead costs. Therefore, we compared the fuzzing throughput with the various IoT fuzzing frameworks mentioned in Section V-B.
E. CODE COVERAGE (EQ3)
Achieving high code coverage in fuzzing can result in mining more vulnerabilities. Hence research on increasing code coverage is crucial. We measured and compared code coverage with the various IoT fuzzing frameworks mentioned in Section V-B.

1) Path coverage. In Fig. 10, we have plotted the cumulative number of unique paths found over time by the FIRM-COV (purple), FIRM-MOptAFL (green), FIRM-AFLFast (red), and the baseline (blue) frameworks for eight different programs. In each plot, the solid line represents the median value from the three rounds. It can be seen that the baseline generally found the same number of paths as some of the other frameworks.

However, it did find marginally more paths than FIRM-MOptAFL in Figs. 10(d) and 10(h). This is because FIRM-MOptAFL’s mutation scheduler did not perform sufficient mutations (including dictionary-based ones) during the first stage. The mutation scheduling configured in FIRM-MOptAFL and FIRM-COV also showed good indicators. For example, in Fig. 10(f), FIRM-MOptAFL found fewer unique paths in the first 4 h than did the other frameworks. However, FIRM-MOptAFL found significantly more unique paths than the others by continuing their mutations during the second stage after 4 h.

Previous research [28] demonstrated that mutation scheduling in AFL is more efficient during the havoc stage (i.e., second stage) than during the deterministic stage (i.e., first stage). We demonstrated this again with our evaluation. As shown, FIRM-AFLFast (where seed-scheduling alone was improved) was effective by itself. However, in Fig. 10(f), improvements in mutation scheduling proved to be more effective than those in seed scheduling. FIRM-COV found the most unique paths in all models compared with other fuzzing frameworks. For example, in Fig. 10(c), FIRM-COV found about 715 % more unique paths than the baseline framework. Thus, we can see that FIRM-COV performs sufficient dictionary-based mutations and discovers various paths by selecting only efficient operators only. As a result, we have shown that our synergy scheduling is efficient.

2) Tuple coverage. AFL collects execution information about the target program while performing fuzzing, storing it in a bitmap. Thus, it efficiently collected statistics on the searched paths. This bitmap includes information on the number of executions of each tuple, and the tuple represents transitions between two specific basic blocks. For example, A->B->C has tuples for AB and BC, and A->B->D has tuples for AB and BD. Thus, there are three unique tuples in both paths. We measured tuple coverage, a different perspective from path coverage. The evaluation results are shown in Table 6. The table shows how much the IoT fuzzing frameworks increased or decreased the number of tuples compared with the baseline.

In FIRM-AFLFast with improved seed scheduling, the number of tuples was generally higher than that of the baseline. However, there was no meaningful change in some models. As a result, FIRM-AFLFast showed an average increase of 18.5 % in all models used for evaluation. With FIRM-MOptAFL with improved mutation scheduling, there were models having fewer tuples than the baseline. The reason for these results is that mutations were not sufficiently performed during the first stage. Interestingly, in the WR940N model, FIRM-MOptAFL had less path coverage than the baseline. However, in tuple coverage, it found 31.6 % more unique tuples than in the baseline. FIRM-MOptAFL did not find various paths in the WR940N model, but we can see that it found multiple tuples for one path. As a result, FIRM-MOptAFL showed an average increase of 9.4 % in all models used for evaluation. In FIRM-COV with introduced synergy scheduling, the number of tuples was generally higher than in the baseline. Moreover, in DIR-815, where the number of tuples increased the most, it increased by 237.7 % compared with the baseline.
J. Kim et al.: FIRM-COV: High-Coverage Greybox Fuzzing for IoT Firmware via Optimized Process Emulation

In summary, in terms of code coverage, we can see that improving seed scheduling is more efficient than improving mutation scheduling and improving both seed and mutation scheduling is most effective. Furthermore, because IoT programs require structured input, we can see that it is rather inefficient if mutations are not sufficiently performed during the deterministic stage, which is the first stage that includes dictionary-based mutation.

F. DISCOVERED VULNERABILITIES (EQ4)

The ultimate purpose of fuzzing is to mine vulnerabilities by monitoring the state of the target. Therefore, we evaluated vulnerability discovery’s effectiveness by fuzzing real-world IoT programs with the various IoT fuzzing frameworks mentioned in Section V-B.

1) 1-day vulnerabilities. We first identified 1-day vulnerabilities for all IoT devices used in the evaluation. Then, we calculated the first crash times that caused the identified vulnerabilities and recorded them in Table 7. We also plotted the cumulative number of unique crashes found over time by the FIRM-COV (purple), FIRM-MOptAFL (green), FIRM-AFLFast (red), and baseline (blue) frameworks in Fig. 11. In each plot, the bar represents the median value from the three rounds.

As shown in the evaluation results, FIRM-COV found crashes related to 1-day vulnerabilities identified in all IoT programs and mined vulnerabilities faster than all other fuzzing frameworks.

As one of the interesting evaluation results, baseline, FIRM-AFLFast, and FIRM-MOptAFL did not find some 1-day vulnerabilities in all IoT programs. In particular, in the WR940N model, they did not find any known 1-day vulnerability. As shown in Fig. 11(h), all the IoT fuzzing frameworks did find crashes, but...
those found in the baseline by the FIRM-AFLFast and FIRM-MOptAFL frameworks were all false positives. This is because they considered any input value that did not respond to the target IoT program to be a crash. In contrast, FIRM-COV discarded the input value that did not respond from the target IoT program. Hence, no false-positives occurred. According to the evaluation results shown in Fig. 11, we can see that after 24 hours, FIRM-COV found the most crashes amongst all IoT programs. An interesting observation is that while FIRM-COV initially found the fewest crashes (as depicted in Fig. 11(c)), over time, it found the most number of crashes amongst all the IoT fuzzing frameworks. As a result, FIRM-COV can find the most unique crashes compared with several IoT fuzzing frameworks. It has the fastest vulnerability discovery, and it discovers all identified 1-day vulnerabilities.

2) **0-day vulnerabilities.** We found two 0-day vulnerabilities using FIRM-COV. We fuzzed two IoT programs with different frameworks in the same environment, but they did not find a crash related to 0-day vulnerabilities.

We reported them to IoT manufacturers; the details of these two 0-day vulnerabilities are as follows:

- Buffer overflow in D-Link DIR-825 (firmware version: 2.02NA to 2.10NA). Attackers could exploit the device by crafting the uniform resource indicator (URI) in httpd.
- Buffer overflow in Trendnet TEW-632BRP (firmware version: 1.10B32). Attackers could exploit the device by crafting the URI in httpd.

3) **Accuracy of vulnerability reports.** We manually performed dynamic analysis for the accuracy of the vulnerabilities discovered by FIRM-COV. The results are shown in Fig. 12. The figure shows the number of reported crashes, including false positives. FIRM-COV generally had high accuracy finding vulnerabilities but had many false-positives with the TV-IP110WN model. These false positives were caused by the firmware handling the process by reading and writing from configuration files in the file system. It broke the configuration file’s entire grammar, owing to previously fed input values, and some subsequent input values were treated as

---

**TABLE 7. Time to crash for 1-day vulnerabilities.**

| Exploit ID   | Model    | Version      | Program | Time To Crash |
|--------------|----------|--------------|---------|---------------|
|              |          |              |         | Baseline | FIRM-AFLFast | FIRM-MOptAFL | FIRM-COV |
| CVE-2016-1558 | DAP-2695 | 1.11RC044    | httpd   | 13 min | 44 min | 1 h 31 min | 7 min   |
| CVE-2017-3195 | DIR-850L | 1.03         | hnap    | 1 h 11 min | 37 min | 1 h 11 min | 8 min   |
| CVE-2017-13772 | WR940N | V4            | httpd   | None | None | None | 3 h 28 min |
| CVE-2018-19241 | TV-IP110WN | 1.2.2.68 | video.cgi | 10 h 6 min | 22 min | 4 h 41 min | 19 min |
| CVE-2019-11418 | TEW-632BRP | 1.10B32 | httpd   | None | 1 h 31 min | 1 h 7 min | 33 min |
| EDB-1235       | DIR-825 | 2.02NA       | httpd   | None | None | None | 1 h 17 min |
| EDB-12366      | DIR-8115 | 1.00B05     | hnap    | 56 min | 33 min | 22 min | 5 min   |
| EDB-123563     | DIR-815 | 1.01         | hedwig.cgi | 6 h 5 min | 5 h 20 min | 1 h 11 min | 49 min |
In this section, we provide information on the limitations that currently exist in the design of FIRM-COV and ideas for potential future research.

1) **Dictionary generation.** FIRM-COV extracts all strings from the target binary, selects the strings used as arguments to the library APIs and includes them in the dictionary. This approach effectively improves the input generation of fuzzing, but functions that manipulate strings are not implemented only with the library APIs. For example, we can implement functions such as string comparison in binary without using the library APIs. In future research, we plan to improve the dictionary generation mechanism for more flexible processing.

2) **Crash verification.** FIRM-COV can detect crashes, such as memory corruption in the target program, but it cannot automatically verify the crashes’ root cause. Because FIRM-COV only aims to mine vulnerabilities in IoT devices, it must be manually verified if the crashes found are vulnerabilities. In future research, we plan to automatically verify the causes of crashes for convenience and accuracy.

3) **Scalability.** The emulator of FIRM-COV supports three architectures: MIPSEL, MIPSEB, and ARM. However, it is essentially based on the FIRMADYNE [4] framework. Hence, it can only emulate Linux-based firmware. Previous research [5] classified IoT devices into three types, and FIRM-COV can analyze only Type-1 devices. In future research, we plan to analyze non-Linux-based IoT devices as well.

**VI. DISCUSSION AND LIMITATIONS**

In this section, we provide information on the limitations that currently exist in the design of FIRM-COV and ideas for potential future research.

**VII. RELATED WORK**

**A. IoT DEVICE SECURITY**

Shoshitaishvili *et al.* [3] proposed Firmalice, which can detect an authentication vulnerability (e.g., backdoor) in firmware. It identifies some of the logic of the firmware related to authentication, and then symbolically executes the firmware with the symbolic execution engine. Thirumalai *et al.* [43] proposed a methodology that implements three-stage encryption and two-stage decryption using the Diophantine equation and RSA public keys to solve the data integrity and authentication problems, which are major security issues for IoT devices. However, other aspects of fuzzing, such as authentication issues, are not specific to our domain and do not affect our research.

Avatar [2] is a dynamic analysis system based on partial emulation that controls the execution of the emulator with the actual embedded device. FIRMADYNE [4] is an automated system capable of emulation and dynamic analysis of Linux-based firmware. It can check for the existence of 60 known vulnerabilities. However, these tools do not use techniques such as fuzzing to find unknown vulnerabilities. In contrast, FIRM-COV can discover unknown vulnerabilities.

Costin *et al.* [44] conducted a large-scale study to detect web interface-related vulnerabilities in embedded firmware, and proposed an emulation framework capable of automated web interface vulnerability detection. Their proposed framework automatically analyzes the web interface within the firmware using both static and dynamic analysis tools. However, because it cannot interact with peripheral devices such as NVRAM, the emulation is unstable. In contrast, FIRM-COV intercepts calls to NVRAM-related functions, which are non-volatile memory, and returns fake data, hence the emulation is stable.

**B. IoT FUZZING**

AFL [20] is a representative tool in the field of fuzzing and is a greybox fuzzer. It can measure code coverage for a target program, support multiple architectures, and discover known and unknown vulnerabilities. However, because some IoT programs have hardware-dependent functions, AFL can lose stability. In contrast, FIRM-COV can perform fuzzing while processing hardware-dependent functions.

Muench *et al.* [5] discussed the challenges of fuzzing for embedded devices and further proposed an emulation-based system capable of detecting fault states. They demonstrated that the full-system emulation-based fuzzer is more effective than the partial emulation-based fuzzer. However, because the proposed system is based on blackbox fuzzing, the code coverage for the target is not considered. In contrast, FIRM-COV is based on greybox fuzzing and attains high code coverage for the target.

Chen *et al.* [7] pointed out the difficulty of acquiring IoT firmware, and proposed IOTFUZZER, which can find memory corruption vulnerabilities without IoT firmware. However, because it is a mobile app-based IoT fuzzing system, it cannot perform fuzzing without mobile apps corresponding to IoT devices. In contrast, FIRM-COV does not rely on mobile apps.

Zheng *et al.* [9] pointed out that the performance of full-system emulation was not ideal and proposed FIRM-AFL.
with high throughput in combination with user-mode emulation. However, they did not study structured input-value generation for IoT programs. In contrast, FIRM-COV can generate structured input values for IoT programs with efficient dictionary generation.

Gu et al. [10] proposed FIRM-CORN, an IoT fuzzing system based on a CPU emulator. Unlike existing full-system emulators, it uses fewer computing resources and optimizes the execution process. However, it requires real-world IoT devices to which to dump the context information. In contrast, FIRM-COV can perform fuzzing without real-world IoT devices.

**VIII. CONCLUSION**

In this paper, we proposed FIRM-COV, an emulation-based IoT fuzzing framework. To achieve high coverage, high throughput, and high compatibility fuzzing, we developed a series of new techniques: a *dictionary generation algorithm* that generates structured input values for IoT programs; *synergy scheduling* that enhances the scheduling of the fuzzer for the efficiency of fuzzing; and *optimized-process emulation* that detects the panic state of the emulator and handles hardware-dependent functions for the accuracy and stability of fuzzing.

We measured the performance overhead of FIRM-COV and evaluated the effectiveness of the developed techniques. We also configured several different IoT fuzzing frameworks, including FIRM-COV, and evaluated real-world IoT programs. The results indicated that FIRM-COV found the most 1-day vulnerabilities and had the highest code coverage of all the IoT fuzzing frameworks. Furthermore, it found two 0-day vulnerabilities.

**REFERENCES**

[1] Cisco Visual Networking Index: Forecast and Trends 2017-2022, Cisco, San Jose, CA, USA, 2018. [Online]. Available: https://cyrekdigital.com/public/uploads/content/files/201-white-paper-c11-741490.pdf

[2] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, “Avatar: A framework to support dynamic security analysis of embedded systems’ firmware,” in Proc. NDSS, vol. 23, 2014, pp. 1–16.

[3] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmaluce-automatic detection of authentication bypass vulnerabilities in binary firmware,” in Proc. Netw. Distrib. Syst. Secur. Symp., 2015, p. 1.

[4] D. D. Chen, M. Egele, M. Woo, and D. Brumley, “Towards automated dynamic analysis for Linux-based embedded firmware,” in Proc. Netw. Distrib. Syst. Secur. Symp., vol. 16, 2016, pp. 1–16.

[5] 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. NDSS, 2018, pp. 1–15.

[6] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, “Avata2: A multi-target orchestration platform,” in Proc. Workshop Binary Anal. Res., vol. 18, 2018, pp. 1–11.

[7] J. Chen, W. Diao, Q. Zhao, C. Zu, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, “iottfuzzer: Discovering memory corruptions in IoT through app-based fuzzing,” in Proc. NDSS, 2018, pp. 1–15.

[8] 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 Secure Privacy Internet Things, 2019, pp. 15–21.

[9] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, “Firm-afi: 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[10] Z. Gui, H. Shu, F. Kang, and X. Xiong, “FIRM-CORN: Vulnerability-oriented fuzzing of IoT firmware via optimized virtual execution,” IEEE Access, vol. 8, pp. 29826–29841, 2020.

[11] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” Commun. ACM, vol. 33, no. 12, pp. 32–44, Dec. 1990, doi: 10.1145/9627.96279.

[12] S. Nagy and M. Hicks, “Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2019, pp. 787–802.

[13] P. Godefroid, A. Kiezun, and M. Y. Levin, “Grammar-based whitebox fuzzing,” in Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement., 2008, pp. 205–215.

[14] M. Eddington. (2011). Peach Fuzzing Platform. [Online]. Available: https://peachtech.github.io/peach-fuzzer-community/

[15] M. Security. (2018). Drarma: A Generation-Based, Context-Free Grammar Fuzzer. [Online]. Available: https://github.com/MozillaSecurity/dharma

[16] J. Johnson. (2018). Gramfuzz. [Online]. Available: https://github.com/d0csavage/gramfuzz

[17] J. Pereyda. (2016). Boofuzz. [Online]. Available: https://github.com/tpereyda/boofuzz

[18] M. Sutton, A. Greene, and P. Amini, Fuzzing: Brute Force Vulnerability Discovery. London, U.K.: Pearson, 2007.

[19] R. Swiecki. (2018). Hongfuzz. [Online]. Available: https://hongfuzz.dev/

[20] M. Zalewski. (2015). American Fuzzy Lop. [Online]. Available: http://lcamtuf.coredump.cx/af

[21] K. Serebryany, “Continuous fuzzing with libFuzzer and AddressSanitizer,” in Proc. IEEE Cybersec. Develop. (SecDev), Nov. 2016, p. 157.

[22] Triforceafl. Accessed: Nov. 19, 2018. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[23] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in Proc. NDSS, vol. 16, 2016, pp. 1–16.

[24] S. Rawat, V. Jain, A. Kumar, L. Cojocar, and C. Giuffrida, “VUzzr: Application-aware evolutionary fuzzing,” in Proc. NDSS, vol. 17, Feb./Mar. 2017, pp. 1–14.

[25] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as Markov chain,” IEEE Trans. Softw. Eng., vol. 45, no. 5, pp. 489–506, May 2019.

[26] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, “Hawkeye: Towards a desired directed grey-box fuzzer,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2018, pp. 2095–2108.

[27] J. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, “QSYM: A practical concolic execution engine tailored for hybrid fuzzing,” in Proc. 27th Secur. Symp., 2018, pp. 745–761.

[28] C. Lu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, “MOPT: Optimized mutation scheduling for fuzzers,” in Proc. 28th Secur. Symp., 2019, pp. 1949–1966.

[29] J. Kim and J. Yun, “Poster: Directed hybrid fuzzing on binary code,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., 2019, pp. 2637–2639.

[30] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-fuzz: Fuzzing by program translation,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2018, pp. 697–710.

[31] F. Bellard, “QEMU, a fast and portable dynamic translator,” in Proc. USENIX Annu. Tech. Conf., vol. 41, 2005, p. 46.

[32] Decaf (Dynamic Executable Code Analysis Framework). Accessed: Aug. 15, 2019. [Online]. Available: https://github.com/decaf-project/DECAF

[33] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, “Repeatable reverse engineering with panda,” in Proc. 5th Program Protection Rev. Workshop, 2015, pp. 1–11.

[34] N. A. Quynh and D. H. Vu, Unicorn: Next Generation CPU Emulator Framework. Las Vegas, NV, USA: BlackHat USA, 2015.

[35] M. Charlie, “Fuzz by number,” in Proc. CanSecWest Conf., 2008. [Online]. Available: https://www.west.net/csw08/csw08-miller.pdf

[36] Firmware Analysis Tool. Accessed: Oct. 14, 2019. [Online]. Available: https://github.com/ReFirmLabs/binwalk

[37] E. Andersen. Uclibc-Official Site. Accessed: May 15, 2012. [Online]. Available: https://www.uclibc.org/

[38] C. Lemieux and K. Sen, “FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage,” in Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng., Sep. 2018, pp. 475–485.
HYUNWOOK KIM received the M.S. degree in computer and information security from Sejong University, Seoul, South Korea, in 2021. His research interests include firmware vulnerability detection, the IoT security, and software security.

FAYOZBEK RUSTAMOV received the B.S. degree from the Economics and Management in the Sphere of ICT in 2013 and the M.S. degree in information security, cryptography, and cryptanalysis from the Tashkent University of Information Technology (TU11T) in 2015. He is currently pursuing the Ph.D. degree in computer and information security with Sejong University, Seoul, South Korea. His research interests include the software security and vulnerability detection.

JOOBEOM YUN received the B.S. degree in computer science and engineering from Korea University, Seoul, South Korea, in 1999, the M.S. degree in computer engineering from Seoul National University, Seoul, in 2001, and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2012. He is currently an Associate Professor with the Department of Computer and Information Security, Sejong University, Seoul. His research interests include software security, artificial intelligence (AI) security, and network security.

* * *

JUHWAN KIM received the M.S. degree in computer and information security from Sejong University, Seoul, South Korea, in 2019, where he is currently pursuing the Ph.D. degree in computer and information security, and convergence engineering for intelligent drone. His research interests include fuzzing, vulnerability detection, the IoT security, and software security.

JIHYEON YU is currently pursuing the M.S. degree in computer and information security, and convergence engineering for intelligent drone with Sejong University, Seoul, South Korea. His research interests include fuzzing, vulnerability detection, the IoT security, and AI security.

[39] *Ida-Official Site*. Accessed: Apr. 13, 2015. [Online]. Available: https://www.hex-rays.com/products/ida/

[40] *Idapython*. Accessed: Apr. 21, 2015. [Online]. Available: https://github.com/idapython/src

[41] *Firmadyne Datasheet*. Accessed: Feb. 21, 2016. [Online]. Available: https://cmu.app.boxcn.net/s/hnpvf1n72acnhysf307cc2nb99fxxmpj

[42] *Nbench*. Accessed: Jan. 2, 2017. [Online]. Available: https://fossies.org/linux/misc/old/nbench-byte-2.2.3.tar.gz/

[43] C. Thirumalai, S. Mohan, and G. Srivastava, “An efficient public key secure scheme for cloud and iot security,” *Comput. Commun.*, vol. 150, pp. 634–643, May 2020.

[44] 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.