Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.035835
Article

Adaptive Emulation Framework for Multi-Architecture IoT Firmware Testing

Jihyeon Yu[1], Juhwan Kim[1], Youngwoo Lee[1], Fayozbek Rustamov[2] and Joobeom Yun[1,*]

[1]Department of Computer and Information Security, and Convergence Engineering for Intelligent Drone, Sejong University, Seoul, 05006, Korea
[2]Department of Computer and Information Security, Sejong University, Seoul, 05006, Korea
*Corresponding Author: Joobeom Yun. Email: jbyun@sejong.ac.kr
Received: 06 September 2022; Accepted: 07 January 2023

Abstract: Internet of things (IoT) devices are being increasingly used in numerous areas. However, the low priority on security and various IoT types have made these devices vulnerable to attacks. To prevent this, recent studies have analyzed firmware in an emulation environment that does not require actual devices and is efficient for repeated experiments. However, these studies focused only on major firmware architectures and rarely considered exotic firmware. In addition, because of the diversity of firmware, the emulation success rate is not high in terms of large-scale analyses. In this study, we propose the adaptive emulation framework for multi-architecture (AEMA). In the field of automated emulation frameworks for IoT firmware testing, AEMA considers the following issues: (1) limited compatibility for exotic firmware architectures, (2) emulation instability when configuring an automated environment, and (3) shallow testing range resulting from structured inputs. To tackle these problems, AEMA can emulate not only major firmware architectures but also exotic firmware architectures not previously considered, such as Xtensa, ColdFire, and reduced instruction set computer (RISC) version five, by implementing a minority emulator. Moreover, we applied the emulation arbitration technique and input keyword extraction technique for emulation stability and efficient test case generation. We compared AEMA with other existing frameworks in terms of emulation success rates and fuzz testing. As a result, AEMA succeeded in emulating 864 out of 1,083 overall experimental firmware and detected vulnerabilities at least twice as fast as the experimental group. Furthermore, AEMA found a 0-day vulnerability in real-world IoT devices within 24 h.

Keywords: Internet of things (IoT); emulation framework; firmware; fuzzing; concolic execution; vulnerability

1 Introduction

Internet of things (IoT) devices are increasingly used in most aspects of our lives. They are also used in sensitive areas such as autonomous driving and the medical industry. According to a Grand

View research report [1], the global IoT market size was valued at approximately $161 billion in 2018, and the revenue forecast for 2025 was estimated to be $950 billion. However, the security of IoT devices remains at a relatively low level because of the huge competition between manufacturers with regard to technologies. This poor sense of security and various targets are bound to be optimal for attackers to exploit. For example, in early 2022, Mozi botnet [2], a malicious code, was found in 11,700 IoT devices such as wireless routers, closed-circuit televisions (CCTV), and advertising monitors. Mozi botnet spreads itself, and the statistic is based on attempts to attack public institutions. Therefore, it is predicted that there will be more actual infected devices.

To prevent these threats, researchers have recently focused on automated firmware testing for the security of IoT devices. In particular, studies have proposed methods that run firmware in a virtual emulation environment and then applied testing techniques, considering the diversity of IoT devices. These methods are practical for large-scale firmware analysis because they can operate various firmware with fewer resources and without actual devices. In addition, because considerable firmware is similar to existing software, traditional software testing techniques can be easily applied to the firmware inside the emulation environment. Thus, many emulation frameworks [3–9] have demonstrated their practicality by combining the fuzzing technique that is effective in finding unexpected vulnerabilities and is capable of automatic analysis. Therefore, we designed and implemented an emulation framework that combines fuzzing as a testing technique.

However, the emulation framework for IoT firmware testing has several limitations. First, existing emulation frameworks lack support for various firmware architectures that are used in IoT-enabled devices to optimize their resources and performance. For example, ThreadX [10] and real-time executive for multiprocessor systems (RTEMS) [11], one of the real-time operating systems (RTOS) in IoT devices, can be built with more than 10 architectures except for a microprocessor without interlocked pipeline stages (MIPS) and advanced RISC machine (ARM), which are major firmware architectures. However, firmware emulation studies have only focused on MIPS and ARM architectures. Firm-AFL [3], an augmented process emulation framework, can only emulate MIPS and ARM architectures, and P2IM [5] is an emulation framework specialized for modeling peripherals with support only for the analysis of ARM series architectures. Quick emulator (QEMU) [12] supports the emulation of some exotic firmware architectures, but operating QEMU [12] is a challenge because it supports manual emulation, not automation. Second, the emulation success rate is low because it is difficult to accommodate many firmware types and versions. In a study on large-scale analysis, FIRMADYNE [13] achieved an emulation success rate of only approximately 16%. This problem cannot be solved unless the functions of all physical devices are fully emulated. Third, most IoT programs should receive inputs that construct a fixed data format such as a packet. In Firm-AFL [3], keywords that reference values used for fuzzing have been collected using a manual analysis and subsequently passed to the fuzzer. This approach is expensive and requires considerable human resources. Therefore, automatically generating structured inputs is necessary when testing new firmware.

To overcome the above limitations, we propose the adaptive emulation framework for multi-architecture IoT firmware testing (AEMA). The objectives of our study are as follows: (1) High compatibility: The proposed system is an adaptive emulation framework for major and minor firmware architectures. We implemented a minority emulator that includes automated configuration, interconnection, and crash handler modules for exotic firmware. (2) High availability: Our system can access target programs with a success rate of 79% from the experimental firmware without physical devices. (3) High practicality: AEMA can achieve higher coverage and detect more crashes than existing studies. AEMA can emulate exotic firmware architectures such as Xtensa, performance optimization with enhanced RISC-performance computing (PowerPC), and ColdFire, through its

minority emulator. Furthermore, we applied arbitration techniques in augmented process emulation to increase emulation availability. In addition, we implemented a keyword extraction technique using concolic execution to refer to the structured input of the firmware target program as a test case for the fuzzing step. As a result, AEMA is less affected by the target firmware and the type of firmware architecture and can test the target firmware widely and quickly based on the extracted keywords.

We compared AEMA with other IoT emulation frameworks and evaluated the emulation stability of 1,083 real-world IoT firmware from various vendors. In addition, the fuzzing results for 10 real-world IoT firmware were compared. AEMA showed an improvement in the emulation success rate by more than 62% compared with other frameworks. The adaptive emulation could emulate various firmware architectures. The keyword extraction technique achieved the highest path coverage compared with other frameworks for IoT testing. As a result, AEMA detected meaningful crashes the fastest and found a 0-day vulnerability.

Based on the challenges referred to above, we summarize the contributions of our study as follows:

1. We integrated an augmented process emulation with arbitration techniques that mitigate the causes of five major emulation failures to achieve high emulation success rates.
2. We propose a novel minority emulator to accommodate exotic firmware to an automated emulation framework. A minority emulator enables analysis of various firmware by emulating firmware architectures, such as Xtensa, PowerPC, RISC version five (RISC-V), and ColdFire, which were rarely addressed in previous studies.
3. We propose a keyword extraction technique to perform a static analysis for efficient test case generation using concolic execution. AEMA with the keyword extraction technique showed several times faster detection and identified more vulnerabilities than the experimental group.
4. We designed and implemented AEMA, an adaptive emulation framework for multi-architecture IoT firmware testing, for the first time. In various aspects of experiments, we showed that AEMA outperforms previous frameworks in testing IoT firmware.

The rest of this paper is organized as follows. We detail the background of IoT firmware testing in Section 2. Next, we explain our methodology for emulation and fuzzing in Section 3. We then present the evaluation compared with the previous studies in Section 4. We discuss this study's limitations and future works in Section 5. We then introduce related work on emulation and IoT fuzzing in Section 6. Finally, we summarize our system in Section 7.

2 Background

2.1 Types of IoT Devices

IoT devices can be categorized according to their purpose and configuration. Classifying IoT device types in firmware analysis is essential for configuring compatible emulation environments. Muench et al. [14] and Wright et al. [15] classified the types of IoT devices into three categories, and the description is as follows.

2.1.1 General-Purpose Embedded Systems (Type-1)

General-purpose embedded systems are mounted on devices by reducing and optimizing operating systems (OS) used in existing desktops. These systems can also be called type-1 embedded systems. The components of type-1 IoT devices are displayed in Fig. 1. These systems maintain applications in the form of file systems, such as the desktop OS, but they have a configuration that lightens OS

commands with a *busybox* and the C library with *uclibc*. For firmware testing of type-1 embedded systems, applications that can be input/output, such as *busybox* or *httpd*, are primarily targeted. Examples of general-purpose embedded systems include Linux-based IoT devices and embedded Windows OS. Although many related studies [3,4,8,9] have focused on these systems because of their similarity with existing desktop programs, several challenges remain such as emulation instability and specific input generation for testing.

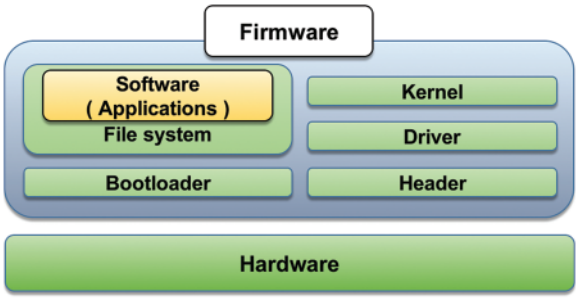

Figure 1: Components of general-purpose IoT devices

2.1.2 Special-Purpose Embedded Systems (Type-2)

Special-purpose embedded systems also have an OS. However, this OS is not universal and is specifically developed for IoT devices. These systems can be referred to as type-2 embedded systems. As these systems are designed for a specific goal, such as flying a drone and CCTV monitoring, the embedded system's configuration is optimized for each device's purposes. An unfamiliar OS renders it challenging to apply analysis techniques to an existing desktop OS. These systems include some RTOSs [16], ZephyrOS [17], and VxWorks [18].

2.1.3 Bare-Metal Embedded Systems (Type-3)

Bare-metal embedded systems do not have an OS or only have OS libraries. These systems can be referred to as type-3 embedded systems. These systems generally consist of a single binary and control hardware directly. Among the three types of embedded systems, it is the lightest and most widely used to construct a single-purpose embedded system. Therefore, some firmware analysis studies [5,19] have focused on type-3 embedded systems. However, analyzing a single binary without an OS has some inherent problems such as interface identification, various architectures, and peripheral access.

2.2 IoT Firmware Emulation

IoT devices perform specified tasks within limited environments and resources. IoT devices typically consist of software, hardware, and firmware that allow them to interact. Because the firmware is responsible for controlling IoT devices, remote attackers target the firmware. Despite these threats, manufacturers focus on development before product launch and do not disclose firmware information after product launch; therefore, in-depth security testing by analysts is not carried out. Furthermore, directly connecting to the firmware inside actual IoT devices limits large-scale analysis. Thus, universal firmware analysis should be performed in an emulation environment that does not require actual devices. Emulation techniques are categorized into four categories:

2.2.1 User-Mode Emulation

User-mode emulation can launch processes compiled for one central processing unit (CPU) on another at a high execution speed because of the fast system-call translation. For example, in IoT fuzzing, user-mode emulation detects the entry point of the target program to prepare the fuzzing starting point. QEMU [12] includes this technique and features a portable operating system interface (POSIX) signal handling and threading. However, user-mode emulation does not emulate peripheral application programming interfaces (API). Thus, user-mode emulation is faster than other emulations but not a sophisticated emulation.

2.2.2 System-Mode Emulation

This method is also called full-system emulation because it can emulate the entire scope of the firmware with a configurable CPU, memory, and peripherals. The main purpose of this emulation is to fully match the emulation environment to the actual physical device for emulation stability. Representative tools that include system-mode emulation are QEMU [12] and FIRMADYNE [13]. QEMU [12] is a pioneer that is used as the base for other system-mode emulation frameworks frequently and can be used with the target kernel. FIRMADYNE [13] is an automated full-system emulation framework that can detect vulnerability using dynamic analysis. However, the emulation success rate is low because accommodating many types of firmware from various vendors is difficult. This problem cannot be solved unless the functions of all physical devices are fully emulated.

2.2.3 Augmented Process Emulation

This emulation method was proposed by Zheng et al. [3], and the main objective of this emulation is to accommodate the stability of the system-mode emulation and the high throughput of the user-mode emulation. This emulation typically runs the firmware in a user-mode emulator and then switches to a system-mode emulator only when unprocessable system calls occur. However, augmented process emulation has problems with the system-mode emulation's low success rates and input processing based on less-handled QEMU [12] such as inappropriate fuzzing triggers using a specific system call.

2.2.4 Partial Emulation

The partial emulation only emulates the CPU and dumps the full context from an actual IoT device to construct an initial environment. The objective of this emulation is to increase emulation success rates while being lighter than QEMU [12]. FIRMCORN [4] was proposed by adopting a partial emulation and combining it with the unicorn engine [20]. However, this emulation method requires actual IoT devices, and therefore an apparent universal limitation exists for large-scale analysis.

In summary, each emulation technique has clear problems in fully automated analysis. In this study, we utilized the structure of the augmented process emulation for our system. To improve the limitation of this emulation, we applied heuristic configurations to optimize the emulation environment through several interventions rather than pursuing the same environment as the physical devices.

2.3 Fuzzing

Fuzzing is one of the techniques that has achieved successful vulnerability detection in conventional software testing. The fuzzer repeatedly sends generated inputs based on a particular rule or randomly mutated inputs from a host to the target program and monitors the response to detect

unexpected errors. This technique can be categorized into black-box fuzzing, white-box fuzzing, and grey-box fuzzing, relying on the information of the target program.

2.3.1 Black-Box Fuzzing

In black-box fuzzing, the status of a program is not monitored. Only the input and output values are monitored. This method, also called data-driven testing, was primarily used before the introduction of sophisticated techniques to observe a program's internal status. KiF [21], PULSAR [22], Boofuzz [23], and Blendfuzz [24] adopt this technique. Black-box fuzzing exhibits a high fuzzing throughput because no instrumentation or monitoring technique is required. However, it is dependent on fortuity because of the lack of detailed feedback such as the path coverage.

2.3.2 White-Box Fuzzing

In white-box fuzzing, the internal structure of the target program is identified before the fuzzer executes, and test cases are generated based on the information obtained during the target program running. This technique is more systematic for the static analysis of a target program than other techniques and is accessible from multiple methods. Typical white-box fuzzers are scalable automated guided execution (SAGE) [25], FuSeBMC [26], T-fuzz [27], Driller [28], and DrillerGo [29]. White-box fuzzing can evaluate all possible paths inside the program, but it has a high overhead.

2.3.3 Grey-Box Fuzzing

In grey-box fuzzing, the instrumentation utilizes partial information of the target program. Grey-box fuzzers are typically used when a program's scope cannot be determined, such as in the absence of a source code. Limited execution-related information such as static analysis information regarding the program or path coverage is sufficient. American fuzzy lop (AFL) [30], Angora [31], Hawkeye [32], Superion [33], and VUzzer [34] adopt this technique. Because grey-box fuzzing is a compromise between white-box fuzzing and black-box fuzzing, its fuzzing throughput and overhead are intermediate levels among the fuzzing types.

Based on how test cases are generated, fuzzing techniques can also be classified as generation-based or mutation-based. First, in generation-based fuzzing, details, such as the target's format and protocol, are known. Generation-based fuzzers do not require any interesting test cases. These fuzzers can investigate all predefined input formats. However, they must manually implement the target program's configuration, which is time-consuming. Mutation-based fuzzing generates test cases based on other inputs and adds anomalies to previous inputs. Mutation-based fuzzers do not require knowledge of the target program but, instead, transform their input values and learn the input format of the program as coverage. However, mutation-based fuzzers generally require a long time to understand the input of the target program.

In this study, our system focused on grey-box fuzzing and mutation-based fuzzing, considering firmware's various formats and availability. Furthermore, we apply concolic execution techniques for efficient input generation to overcome the drawbacks of mutation-based fuzzing.

2.4 Concolic Execution

Concolic execution is a hybrid software testing technique in which symbolic execution and concrete execution techniques are combined. In concrete execution, all path conditions are collected by running a program with a specified input value. Because a fixed input is provided, the path and result are identical irrespective of the number of runs. Conversely, symbolic execution assumes the path

condition using symbolic values rather than specific values. In symbolic execution, path constraints that were not reached in previous runs are collected through repetitive runs. These constraints are solved by the solver to increase the path coverage. As the number of paths that can be executed increases with the program's size, symbolic execution has an inherent path explosion problem. A concolic execution technique was proposed to mitigate the limitations of each technique. This technique was first proposed in [35]. It explores the program's paths through concrete values and collects unexplored paths as symbolic path conditions to be solved. Subsequently, the collected constraints on the path condition are solved by the constraint solver to obtain new concrete input values. Finally, this operation is executed again using the newly found concrete input values. Concolic unit testing engine (CUTE) [36], KLEE [37], Angr [38], Triton [39], and QSYM [40] adopt this technique. This technique is specifically for solving complex or hard-coded constraint values but may become immersed in an infinite loop during execution, which hinders interesting path exploration. Furthermore, reaching a path containing encryption elements creates conditions that cannot be resolved typically.

2.5 IoT Firmware Analysis Challenges

We identified the main challenges from our empirical experiment and existing IoT firmware analysis studies. The following challenges should be resolved for automated IoT device testing:

2.5.1 Compatibility for Firmware Architectures

Because MIPS and ARM are market-dominant firmware architectures, firmware analysis has focused on these architectures. However, despite the trend of using various firmware architectures for resource optimization in major IoT devices, such as drones, robots, and consoles, research on the automated emulation of non-mainstream firmware architectures is limited. Architectures, such as Xtensa, are capable of emulation implementation in QEMU [12], but studies on automation are insufficient. Therefore, emulation and analyses for exotic firmware architecture had to be manually performed. To achieve large-scale firmware testing, the challenge is to implement a universal emulation framework that can be used for various firmware architectures.

2.5.2 Automated Emulation Failure

Hardwareless emulation requires firmware extraction, architecture identification, network identification and configurations, bootloader setup, and memory layout configurations. Automated universal emulation for various firmware is a challenge because configurations for each firmware are different. Existing studies have several problems such as using incorrect settings collected from the firmware or missing emulation configurations. This problem results in a low success rate in existing automated emulation, such as in FIRMADYNE [13], which exhibits an emulation success rate of approximately 16%. Therefore, automated emulation configurations should be optimized to achieve high success rates for emulation.

2.5.3 Inputs of a Specific Format

Many networks-based IoT devices communicate with a fixed input and output format such as hypertext transfer protocol (HTTP)-based packet format. Furthermore, these formats have countless types of headers and methods. In Firm-AFL [3], format-related keywords are provided through a manual analysis of each example firmware. This method is influenced by the analyst's ability and becomes less effective as the number of targets increases. However, if we perform random fuzzing without passing format-related keywords to the fuzzer, the fuzzer may fail to navigate the deep paths

of the program. As a result, the fuzzer will require considerable time to escape the branch at which the input is rejected.

3 Methodology

In this section, we illustrate the methodology of AEMA. The architecture of AEMA is depicted in Fig. 2, and we explain our system by dividing it into two subsections: adaptive emulation and fuzz testing. Then, we describe our framework implementation.

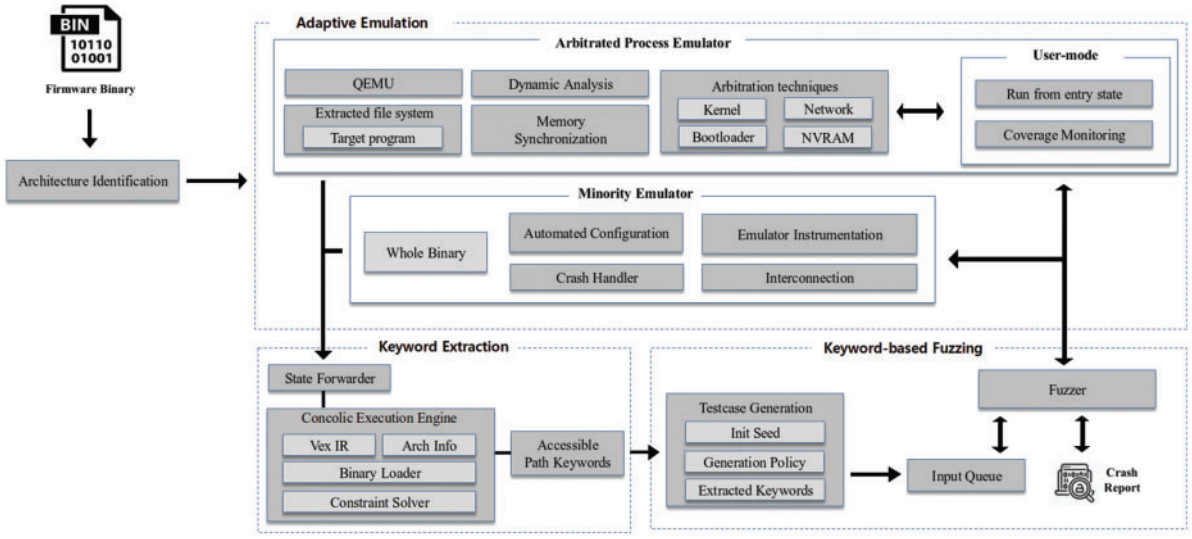

Figure 2: Overview of AEMA

3.1 Adaptive Emulation

In this subsection, we describe the adaptive emulation step of our system. This step includes architecture identification and emulator selection. If the arbitrated process emulation is selected for major firmware, an arbitration technique is performed, and if the minority emulator is selected for exotic firmware, the automated configuration module is performed to establish an environment in which the target firmware can be operated.

3.1.1 Architecture Identification

In the adaptive emulation step, an emulator is selected according to the architecture of the firmware binary. Binwalk [41], readelf, Rohleder [42], and Firmware-mod-kit [43] were used to identify as many firmware types as possible. File system extraction functions for general-purpose IoT firmware are also included in this module. We designed firmware binaries for major architecture to be mapped to the arbitrated process emulator and binaries for exotic architecture to the minority emulator.

3.1.2 Emulations for Multiple Architectures

As mentioned in Section 1, the availability of exotic firmware is also increasing as IoT devices become diverse. Nevertheless, only a few manual emulations and analysis methodologies have been proposed for exotic firmware architecture. Therefore, our goal was to implement an automated emulation framework and expand the scale of firmware analysis by co-existing a minority emulator

for exotic firmware with an improved emulator for major firmware. This technique is called adaptive emulation because the operated emulator is determined according to the firmware architecture. Compatible architectures for the adaptive emulation are presented in Table 1, and we describe each emulator as follows:

Table 1: Compatible architectures of the adaptive emulation determined by firmware

Emulation type	Firmware architecture
Arbitrated process emulator	X86, ARM, ARM hard float (ARMhf), MIPS
Minority emulator	Xtensa, PowerPC, RISC-V, ColdFire

Arbitrated Process Emulation for Major Firmware Architectures: Inspired by a previous study [3], we used an augmented process emulation structure for major firmware architectures such as MIPS and ARM. This approach mitigated the hardware dependency problem of user-mode emulation and the performance overhead of system-mode emulation. However, an augmented process emulation also includes system-mode emulation that is not easy to generalize because emulation configurations, such as network, non-volatile random access memory (NVRAM), and the bootloader, should be accurately defined for various target firmware. To address this problem and achieve high emulation success rates, we propose the arbitrated process emulation of IoT firmware. We applied arbitration techniques to the system-mode emulator in the augmented process emulation. This work compromises the challenge of emulating exactly the same actual device through heuristic methods. As claimed by [44] and [45], the heuristic tuning of the configuration increases the emulation success rate without a considerable effect on the firmware execution flow. Details of the arbitration techniques are explained in the next section.

Minority Emulator for Exotic Firmware Architectures: Many IoT firmware do not have a response or logging function for memory corruption because IoT devices have environments with minimal resources. This limited feedback may result in missing the crash or a stuck state during fuzz testing. It eventually leads to poor fuzzing performance. Therefore, the core function of the minority emulator is the interaction between the emulator and the outside. We implemented the interconnection module that acts as a bridge between the fuzzer and emulator to perform crash handling in the middle. An overview of the minority emulator is shown in Fig. 3. First, the fuzzer runs the interconnection module that delivers mutated inputs and maintains the fuzzer configuration to the minority emulator through a socket. Then, the fuzzer sends mutated inputs, and the minority emulator listens on a socket and waits for inputs from the interconnection module. If the minority emulator receives input data, then the emulation will be executed. The executed block addresses are instrumented and sent to the fuzzer as the path coverage. Finally, the minority emulator reports the final execution status to the interconnection module. If a crash state is detected, then the interconnection module sends a segfault signal to the fuzzer in terms of crash handling. Inspired by the work of Muench et al. [14], we implemented a separate crash handling module in AEMA that includes plugins related to memory corruption. Furthermore, we set the response timeout to the default 1 min to escape the stuck state during fuzzing. Emulation initialization was designed to ensure that it is automatically configured using the fork-point function for fuzzing restarting. Finally, the data register that acts as an input parameter empirically specifies the most used register for each architecture. Accordingly, the fork point at which the fuzzing starts is the address at which the specified data register was first used in the emulator.

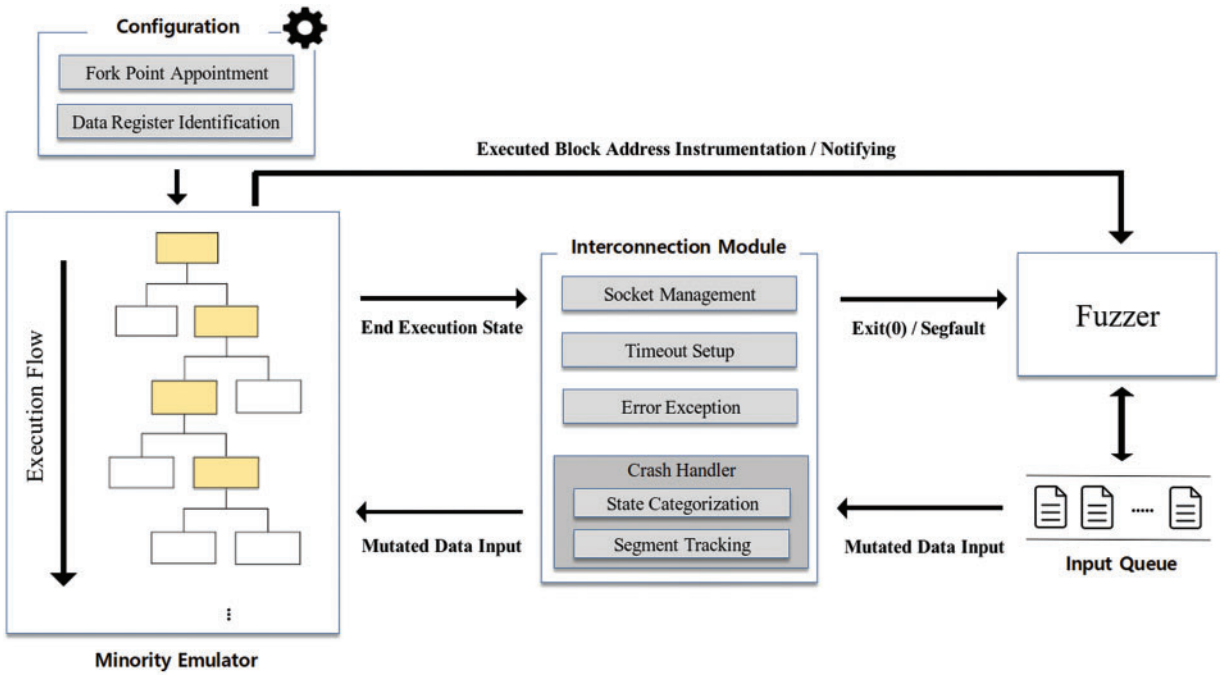

Figure 3: Workflow of the minority emulator

3.1.3 Arbitration for Emulation Configuration

Existing emulation framework studies [12,13] have proposed to ensure that the emulator fully mimics the actual operation of the target firmware. However, as mentioned before, considering the diversity of firmware, applying these results to a large-scale system is impossible. To address this problem, we applied arbitration techniques to the augmented process emulation. Arbitration techniques were first proposed in [45] and focused on emulation availability rather than a perfect reenactment by analyzing the leading causes of failure at each emulation step and performing heuristic modifying firmware execution. The primary causes of existing firmware emulation failures are described below:

Boot: When the firmware starts running in an emulation environment, numerous problems, such as inappropriate booting sequence and inability to recognize the file system, can occur. This may result in a kernel panic. These problems were resolved by enabling an additional initialization file and boot file identification.

Network: When booting is normally completed, the network is configured. As the network enables communications between the analysis system and the emulator, it is a significant component that should be emulated for testing. However, some problems exist in the network configuration such as the lack of network information, invalid internet protocol alias, and multiple network interfaces. These problems were resolved by configuring the network forcibly and setting up only one ethernet interface.

Non-volatile random access memory (NVRAM): An NVRAM is a flash memory that is widely used as a peripheral to store configuration data in IoT devices. FIRMADYNE [13] creates a separate library to support NVRAM. However, NVRAM-related problems, such as the absence of NVRAM default files and only custom default files, still occur. These problems were resolved by finding a file

of the key-value type that is the format of an NVRAM file and by creating an artificial response as if an NVRAM file exists.

Kernel: As in a conventional OS, the firmware has kernels connected to peripherals to control peripheral devices using an *ioctl* command. However, unsupported kernel and version incompatibility problems occur because the kernel is as diverse as the firmware. These problems were resolved by creating an artificial *ioctl* command, compiling a new kernel, and adding compilation options.

Others: Other problems include unexecuted webservers, emulating timeout, and inadequate programs for emulation. These problems were resolved by executing emulation forcibly after searching the web server with corresponding configuration files, optimizing timeout, and adding the latest version of *busybox* into the emulator.

Fig. 4. shows the execution flow of the arbitration technique in our emulator. To reproduce the firmware operation, which is the original purpose of the emulation, our method has the same driving steps as the actual IoT device. In the case of conventional system-mode emulation, if this first emulation fails, it is terminated as is. However, the arbitrated process emulation collects and analyzes logs related to failure so that the firmware can be operated by modifying or adding elements lacking in the steps related to the cause of failure in the next emulation. There may be several modifications in one step, and there may also be steps that do not require modification. Emulation is repeated until the firmware is normally run inside the emulator. When the successful emulation is finally configured, interactions with the user-mode emulator are prepared.

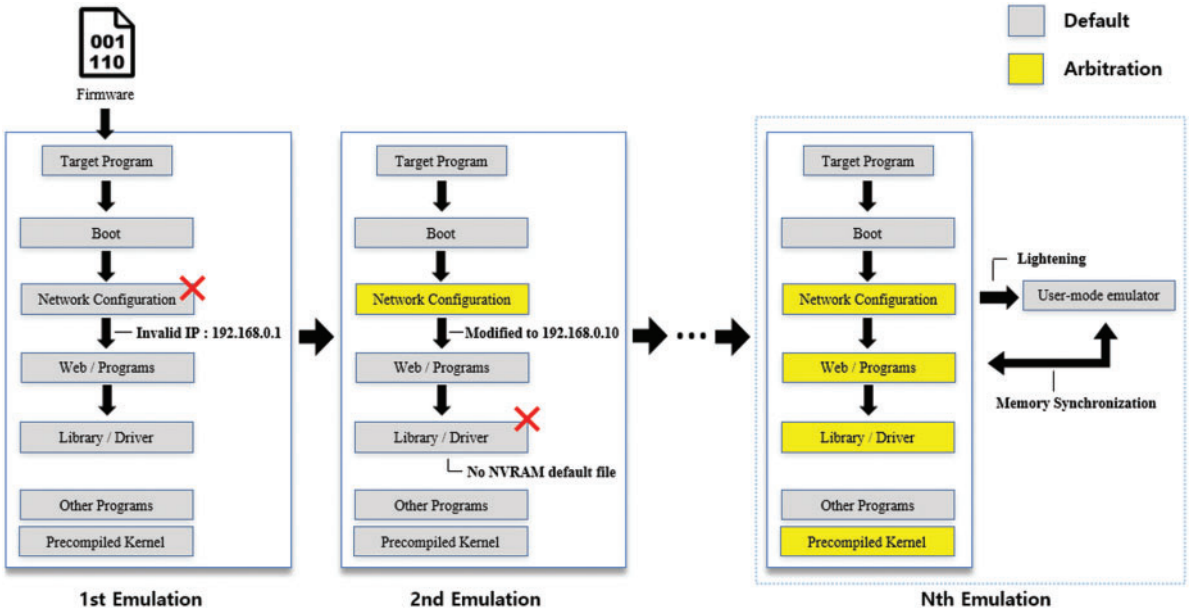

Figure 4: Execution flow of the arbitration technique in the arbitrated process emulator

In the emulation configuration step, arbitrary modification or addition may differ from the operation of the actual device and may concern invalid crash detection in the vulnerability analysis process. However, according to our evaluation in Section 4.2.2, the valid crash detection rates do not significantly decrease compared with that reported in previous studies. In practice, we could obtain high availability at the expense of low accuracy in a large-scale program.

3.2 Fuzzing in the Emulation Environment

In this subsection, we describe the fuzzing step of our system. This step includes the keyword extraction technique and keyword-based fuzzing. The keyword extraction technique is performed through concolic execution, and the collected keywords are transferred to the fuzzer and referred to the test case generation. The fuzzer repeatedly sends the generated test case to the emulator to test for vulnerabilities in the target firmware.

3.2.1 Keyword Extraction

Keywords in the fuzzing technique are used as a key reference when the fuzzer creates test cases, which is essential in firmware fuzzing because the input format is somewhat fixed. The program filters out the input from the fuzzer if IoT fuzzing is performed without any information about the target firmware. We propose the use of a keyword extraction technique utilizing concolic execution to alleviate this problem. Concolic execution is primarily used to solve complex constraints that pass through program branches. Thus, in IoT firmware, we can collect keywords that are accessible and passable to the program branches. An example of path keywords inside the firmware is shown in Fig. 5. The firmware execution paths rely on which header and values the input includes. Therefore, as shown in Fig. 5, a part of the input format is included in each branch such as *cgi* or *chklist.txt*. This fixed input format should be accurately included in the input for efficient fuzzing, but it takes a considerable amount of time for the fuzzer to identify it by itself.

```
__fd = strncmp(__s1 + 1,"cgi",3);
if (((((__fd != 0) && (__fd = strncmp(__s_00,"chklst.txt",10), __
     (__fd = strncmp(__s_00,"wlan.txt",8), __fd != 0)) &&
    ((__fd = strncmp(__s_00,"HNAP1.txt",9), __fd != 0 &&
     (__fd = strncmp(__s_00,"ip_error.txt",0xc), __fd != 0)))) &&
   (((__fd = strncmp(__s_00,"wireless_update.txt",0x13), __fd !=
     ((__fd = strncmp(__s_00,"router_info.xml",0xf), __fd != 0 &&
      (__fd = strncmp(__s_00,"post_login.xml",0xe), __fd != 0))))
```

Figure 5: Path keyword example inside the firmware

Meanwhile, concolic execution collects format-related conditions as symbolic values and creates a formalized constraint for traversing that path. Then, the constraint is solved by a constraint solver engine. However, although concolic execution is lighter than symbolic execution, it is still expensive to explore most branches of the program because it involves symbolic execution and may require solving complex constraints. Therefore, we designed AEMA to collect constraints as a preprocess before fuzzing, considering the performance overhead of concolic execution. The resolved constraints are delivered to the fuzzer before the fuzzing starts and can be expressed as keywords referenced when the fuzzer creates a test case.

Not all the keywords are always meaningful such as repeated constraints in an infinite loop. However, keywords that do not improve the path coverage are excluded from the reference priority by the fuzzer.

3.2.2 State Forwarder

We used Angr [38] as a concolic execution tool for the keyword extraction technique, but existing concolic execution tools, including Angr [38], only support the MIPS and ARM architectures. Thus, we implemented the customized state forwarder using Ghidra [42] to be compatible with some exotic firmware and acquire information that is necessary for concolic execution. Because Ghidra [42] has

numerous supported architectures, such as ARM, MIPS, peripheral interface controller (PIC), and scalable processor architecture (SPARC), some security analyses use Ghidra [42] on firmware-specific problems. Furthermore, Ghidra [42] enables the analyses of exotic firmware architectures that were not originally supported through additional implementations. We utilized Ghidra [42] to implement the state forwarder for extracting information about Xtensa, PowerPC, RISC-V, and Coldfire firmware architectures.

In our system, the state forwarder dumps and stores the information, such as the segment, function, memory layout, and register value, required for concolic execution. The state forwarder creates an initial state at the entry point of the target based on the stored information, enabling concolic execution normally. Then, concolic execution runs and collects constraint values for the entire program path. As some of the collected values also contain filtering branches for the input, the program can be explored comprehensively if the collected values are passed to the fuzzer.

3.2.3 Keyword-Based Fuzzing

In IoT firmware testing, intensive programs, such as *httpd* or *cgibin*, can commonly send messages to the target application only when the target application is specified in the input value. The target applications are generally subdivided into *.cgi* and have various components for each IoT device. Therefore, test cases of the fuzzer may access only shallow and meaningless paths for a long time because paths require a fixed value, or the header in the input format is slightly different for each firmware. Thus, understanding a target program before fuzzing starts is important for improving the fuzzing performance. Test cases ending in the first branch and those ending in the last branch have a significant difference in the number of test cases that can be derived.

For efficient fuzzing in AEMA, we investigated the overall program path with concolic execution and extracted as many program path values as possible. An example of the keyword-based fuzzing input is shown in Fig. 6. Because of the feature of the intensive program, the execution path changes depending on the target application. Therefore, the name of the target can be obtained as a path constraint through the keyword extraction technique. Specific values could also be obtained because the path constraints of the program included values and header such as a hard-coded unique identifier (UID) and domain name. When the fuzzer creates test cases, the existing seed inputs are partially referenced to the collected keywords. The test case is placed in the input queue and mutated for the new test case generation if the path coverage of the target program increases because of the generated test case. In the proposed system, AFL [30] is used as a prototype fuzzer engine. However, this can be replaced with another fuzzer to improve mutation efficiency and throughput in future studies.

Figure 6: Example of a keyword-based fuzzing input

3.3 Framework Implementation

We developed AEMA using Python and C languages to interlock each module. Specifically, we implemented the adaptive emulation using QEMU 2.1.0 [12] for major firmware architectures and Ghidra [42] for exotic firmware architectures. Moreover, we created Python scripts using Binwalk [41] and Firmware-mod-kit [43] to automatically identify and separate architectures and operate emulators suitable for the architecture. In the minority emulator, we implemented separate scripts for each architecture because each architecture has different components such as register names and registers to be used as input. Furthermore, we used Angr version 9.0.6 [38], a concolic execution tool, to implement the keyword extraction technique. In the fuzzer, the improved AFL [30] tool was used in the comparison group, but for AEMA, AFL version 2.52b [30] was used for accurate comparison with the previous study [3].

4 Evaluation

The main evaluation questions for AEMA are as follows: 1) are our emulation methodologies practical, 2) is the keyword extraction technique effective for path coverage improvement, 3) is vulnerability detection performance better than previous studies, and 4) are detected crashes valid for exploitability. The desktop we used to evaluate the results includes an Intel i7 processor, 32 GB of random access memory (RAM), and Ubuntu version 18.04.

4.1 Emulation Success Rate

In this subsection, we show that our emulation methodologies are practical. We conducted emulation experiments on a total of 1,083 firmware and compared our systems with existing large-scale IoT analysis frameworks.

4.1.1 Experimental Setup

To evaluate the emulation success rate, we compared AEMA with large-scale IoT analysis frameworks, Firm-AFL [3] and Firmfuzz [8]. We collected firmware for the experiment based on the FirmAE [45] and Firm-AFL [3] datasets, and some major firmware were additionally collected and configured. Xtensa, PowerPC, RISC-V, and ColdFire firmware were grouped into *Exotic FW*, as shown in Fig. 7, because the number of firmware released is few. The criterion for the emulation success is whether the network configuration is inferred to enable communication with the fuzzer or user. Firm-AFL [3] sets the network configuration inference timeout to 1 min as the default, but we set it to 5 min because the emulation could be successfully performed by simply increasing the inference timeout.

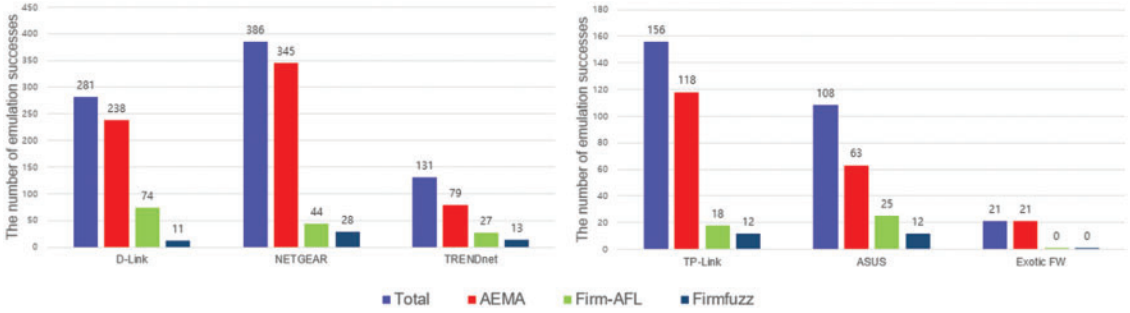

Figure 7: Firmware emulation success rate results for vendors

4.1.2 Experimental Results

Fig. 7 shows the number of successful emulations for each vendor of Firm-AFL [3], Firmfuzz [8], and AEMA. Overall, AEMA exhibited emulation success numbers that are close to the total number of firmware images. In particular, for D-link and NETGEAR, which account for more than half of the total, firmware emulation success rates in AEMA are close to 84% and 89%, respectively, whereas those of other frameworks are less than half that of AEMA. This difference in success rates reveals that verifying the crash detected to the actual device after testing the modified emulation through heuristic techniques is more practical than insisting on emulation reliability, which is an advantage of full-system emulation. In addition, Firm-AFL [3] could not emulate any exotic firmware because unmodified FIRMADYNE [13] provides automated emulation for MIPS and ARM architectures by improving QEMU [12] but does not have components for automated emulation of exotic architectures. Because Firmfuzz is also an emulation framework based on FIRMADYNE [13], the exotic firmware could not be emulated without additional modules.

However, AEMA could emulate all 16 firmware automatically using the minority emulator that included data register identification, memory layout mapping, and hooking function for exotic firmware architectures.

4.2 Fuzzing with AEMA

In this subsection, we show that AEMA is more effective for path coverage and has better crash detection performance than existing frameworks through the keyword extraction technique, the number of crashes found, and time-to-crash. Furthermore, we verify through debugging that the detected crashes are valid for exploitability.

4.2.1 Experimental Setup

As the fuzzing technique includes randomness, the average data must be provided through multiple experiments. In addition, it is better not to test the security test tool for too long because it is disadvantageous in the real world to spend much time on a single test. As in the experimental setting in previous studies [3,4], we performed fuzzing for approximately 24 h for all target programs, and all results are the average of three measurements.

Comparison Group: A comparative group was constructed using the existing IoT fuzzing framework, Firm-AFL [3], to evaluate the fuzzing performance of AEMA. We set the baseline by adding emulation techniques to the Firm-AFL [3] framework for focusing on the fuzzing efficiency evaluation. Furthermore, we constructed a fuzzing framework that includes improved mutation scheduling based on Firm-AFL [3]. They are described below:

(1) *Baseline:* We used Firm-AFL [3] as the baseline for comparative evaluation. However, we added arbitration techniques and the minority emulator to the baseline because some of the firmware in our experimental target is not emulated by Firm-AFL [3].
(2) *Baseline with Search Strategies:* We changed the mutation algorithm of AFL [30] to the power scheduler proposed in [46] for comparison with our system. The power scheduler allocates priorities to low-frequency paths to improve path coverage. Arbitration techniques and the minority emulator were also applied for an accurate fuzz testing evaluation.
(3) *AEMA:* AEMA used an arbitrated process emulator and a minority emulator in this experiment. In addition, AEMA applied the keyword extraction technique to allow the AFL fuzzer [30] to reference the input format of the target firmware when generating test cases.

Target Firmware Selection: Overall, we experimented with the firmware of 10 IoT devices. The summary of the target firmware is presented in Table 2. We chose firmware IDs (1)–(4), which were used by Firm-AFL [3] for comparison with our system. We also selected firmware IDs (5) and (6) that were not evaluated in Firm-AFL [3] for universal evaluation. Finally, IDs (7)–(10) were selected as the exotic firmware to investigate the feasibility of the minority emulator. Major architectures, such as MIPS and ARM, were emulated in the arbitrated process emulation (APE). Exotic architectures, such as Xtensa, PowerPC, RISC-V, and ColdFire, were emulated in a minority emulator (ME).

Table 2: Target firmware summary for the fuzzing experiment

ID	Firmware	Version	Vendor	Device	Emulator	CPU Arch	Description
1	DIR-825	2.02NA	D-Link	Router	APE	MIPSEB	Embedded HTTP Server
2	WR-940N	V4	TPLink	Router	APE	MIPSEB	Embedded HTTP Server
3	DIR-850L	1.03	D-Link	Router	APE	MIPSEB	CGI Binary Program
4	TV-IP110WN	1.2.2.68	Trendnet	Camera	APE	MIPSEB	CGI Binary Program
5	EX-6150	1.0.0.38	Netgear	Wifi extender	APE	MIPSEL	Embedded HTTP Server
6	DNS-327L	1.03B03	D-Link	Router	APE	ARMEL	Embedded HTTP Server
7	ESP-8266	1.5.1	Tensilica	Wifi module	ME	Xtensa (ESP)	Embedded Wifi MCU
8	S1-PPC	–	–	Console	ME	PowerPC	Open Firmware Project
9	S2-RISCV	–	–	Gateway	ME	RISC-V	Open Firmware Project
10	S3-COLDFIRE	–	–	Switcher	ME	ColdFire	Open Firmware Project

4.2.2 Experimental Results

Keyword extraction performance: As exploring deep paths of the program is the same as carefully testing the program, path coverage in fuzzing is an essential evaluation category for vulnerability detection. As mentioned earlier, IoT devices limit inputs that do not fit the program through filtering. This limitation may cause path coverage improvement more inefficient than existing desktop programs. Therefore, we applied keyword extraction techniques using concolic execution because the IoT fuzzer must refer to some format of the target program to generate test cases. Fig. 8 illustrates the path coverage according to the presence or absence of keyword extraction for eight IoT device programs among our experimental groups. All experiment results with the keyword extraction technique revealed that the number of covered paths is up to three times higher than the results without the keyword extraction technique. In addition, all the results showed that the gap in the path coverage increased over time because keywords pushed back in the queue according to the mutation scheduler were used to generate new test cases. However, the results after 12 h exhibited a marginal improvement even when the keyword extraction technique was applied. It is presumed that inefficient inputs were repeated because the extracted keywords were depleted. Another potential cause is that the target program may have complex paths such as recursive function paths. Solving this using concolic execution and fuzzing approaches takes a considerable amount of time.

Crash Detection: The primary purpose of security testing systems, such as fuzzing, is to detect vulnerabilities so that programs can be protected from malicious attackers before and after deployment. Testing real-world IoT device firmware limits time and must be verified for exploitation caused by crashes. Therefore, we evaluated the number of crashes for each IoT program and the time required to discover the first 1-day vulnerability as time-to-crash.

(1) *Number of Crashes found:* We first measured the cumulative number of crashes during the 24 h period in the baseline, the baseline with search strategies, and AEMA, as listed in Table 3. The baseline with search strategies exhibited higher improvement rates than the baseline, and our system was able to detect a high number of crashes than other IoT frameworks except for the TV-IP110WN model. In particular, we were able to identify three times more crashes than the baseline in the DIR-825 and TV-IP110WN models among the major firmware with a large number of crashes found. However, the baseline with search strategies exhibited a higher improvement rate than AEMA in the TV-IP110WN model. It was estimated that the keyword extraction technique was ineffective because of reasons related to resources or the specific structure of the firmware. Nevertheless, the baseline with search strategies generally detected fewer crashes than AEMA because it had no references to structured inputs.

(2) *Time-to-Crash for Exploitability:* We identified 1-day vulnerabilities for real-world firmware and common weakness enumeration (CWE) vulnerabilities for sample firmware in each IoT fuzzing framework. To evaluate the time-to-crash, we have calculated the first crash times that caused the 1-day or CWE vulnerabilities within 24 h and arranged them in Table 4. AEMA discovered vulnerabilities faster than other frameworks except for the TV-IP110WN model. In the TV-IP110WN model, similar to the previous experiment, the baseline with search strategies detected crashes faster than AEMA. As one of the other interesting evaluation results, the baseline and baseline with search strategies could not detect 1-day or CWE vulnerabilities within 24 h in the WR-940N and S1-PPC models, respectively. Thus, all the crashes detected in the WR-940N and S1-PPC models of Table 3 are false positives. It is presumed that no response was received from the input, or faulty emulation configurations were mistaken for a crash. By contrast, during the period of 24 h, AEMA detected 1-day or CWE vulnerabilities. AEMA could find crashes more efficiently than other frameworks, even considering the false positives.

(3) *0-day vulnerability:* We found the 0-day vulnerability using AEMA in approximately 14 h, whereas the baseline with search strategies could not detect it in the same environment within 24 h. We are in the process of registering common vulnerabilities and exposure (CVE), and the details regarding the 0-day vulnerability are as follows.
Buffer overflow in D-Link DIR-850L (firmware version: 1.03). This overflow allows remote attackers to execute an arbitrary code through a crafted HTTP cookie header.

Crash Validation: To evaluate the validity of the crash detected by our system, we manually performed a dynamic analysis of the recorded crashes. The analysis environment was configured where the target program can be observed through remote debugging of the GNU debugger (GDB) in an emulator operating the firmware. Subsequently, we observed memory corruption errors in the target program by transmitting a crash input value. The experimental results of the validation for the recorded crash are displayed in Fig. 9. AEMA exhibited high accuracy and few invalid crashes in the number of recorded crashes. However, the proportion of invalid crashes is slightly higher than that of the false positive experiment of FIRM-COV [9], a firmware fuzzing framework derived from Firm-AFL [3], because the heuristic arbitration techniques were applied to AEMA for the success rate of emulation. Thus, arbitration techniques can cause inconsistencies between the actual device and emulator and some false positive crashes. However, this disadvantage also existed in recent studies of symbolic execution-based emulation frameworks [6,7] because manual emulation is more complicated than manually verifying false-positive crashes. Therefore, we determined that improving the emulation success rate for high availability was a priority over the occurrence of a few invalid crashes. In the future, obtaining the emulation success rate without any tradeoff should be investigated.

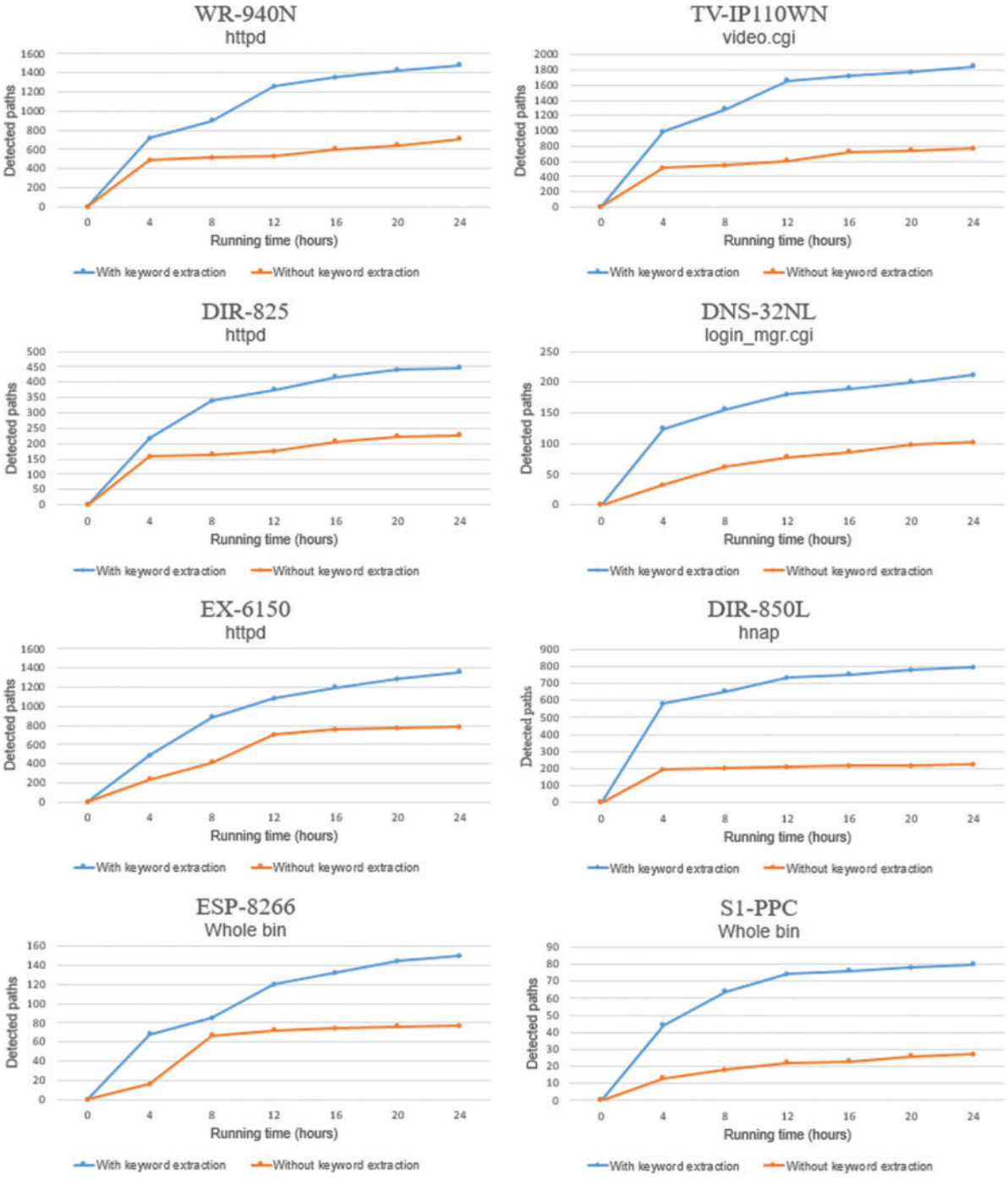

Figure 8: Program path coverage with and without keyword extraction techniques

Table 3: Total crashes found by the baseline, baseline with search strategies, and AEMA on the IoT programs groups for the experiment

Model	Version	Program	Baseline	Baseline with search strategies		AEMA	
			Number of crashes found	Number of crashes found	Increase	Number of crashes found	Increase
DIR-825	2.02NA	httpd	24	52	216%	72	**300%**
WR-940N	V4	httpd	6	7	116%	10	**166%**
DIR-850L	1.03	hnap	147	166	113%	202	**137%**
TV-IP110WN	1.2.2.68	video.cgi	116	525	**452%**	366	315%
EX-6150	1.0.0.38	httpd	52	78	150%	84	**161%**
DNS-327L	1.03B03	login_mgr.cgi	34	61	179%	78	**229%**
ESP-8266	1.5.1	whole bin	2	5	250%	7	**350%**
S1-PPC	–	whole bin	5	5	100%	6	**120%**
S2-RISCV	–	whole bin	4	7	175%	9	**225%**
S3-COLDFIRE	–	whole bin	1	1	100%	4	**400%**

Table 4: Time-to-crash: 1-day vulnerabilities by the baseline, baseline with search strategies, and AEMA on target programs

Exploit ID	Model	Version	Program	Time-to-crash		
				Baseline	Baseline with search strategies	AEMA
EBD-ID-38718	DIR-825	2.02NA	httpd	21 h 47 m	19 h 24 m	**1 h 8 m**
CVE-2017-13772	WR-940N	V4	httpd	*None*	*None*	**6 h 32 m**
CVE-2017-3193	DIR-850L	1.03	hnap	3 h 6 m	42 m 38 s	**38 m 16 s**
CVE-2018-19241	TV-IP110WN	1.2.2.68	video.cgi	10 h 14 m	**4 h 41 m**	8 h 56 m
CVE-2019-20733	EX-6150	1.0.0.38	httpd	4 h 18 m	1 h 6 m	**24 m**
CVE-2014-7859	DNS-327L	1.03B03	login_mgr.cgi	3 h 2 m	22 m	**19 m**
CWE-787	S1-PPC	–	whole bin	*None*	*None*	**18 h 32 m**
CWE-129	S2-RISCV	–	whole bin	2 h 58 m	1 h 36 m	**1 h 2 m**

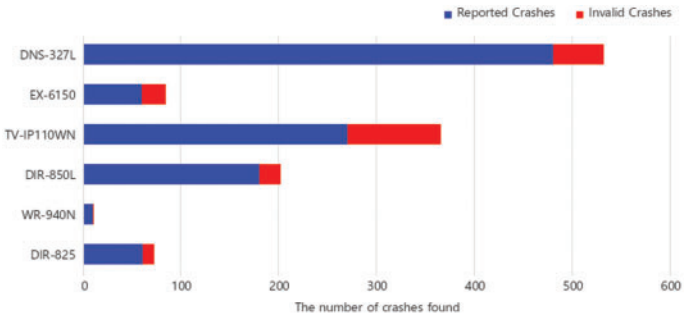

Figure 9: Validation of the effectiveness of reported crashes in AEMA

5 Discussion

In this section, we discuss the remaining limitations of AEMA and the directions for future research.

5.1 Overhead of the Keyword Extraction Technique

In the keyword extraction technique, the concolic execution is used to extract keywords for all the program paths. Therefore, this technique may cause the path explosion, a chronic problem of concolic execution. To bypass this problem, we used the methods that place time constraints on the search time and skip to the next path constraints. However, this method may incur problems such as not solving path constraints that can be solved or extracting overlapping keywords. Therefore, in future works, we plan to improve the keyword extraction technique by applying methods such as slicing paths via specific rules [47] or selective execution technique [48,49].

5.2 Verification for Emulation Reliability

As shown earlier, arbitration techniques improved the emulation success rate using heuristic methods which facilitated dynamic firmware analysis. However, this result may overlook the concept of full-system mode emulation that should be consistent with actual devices for firmware availability. To alleviate these concerns, we will study automated verification techniques for emulation reliability.

5.3 Manual Configuration of the Minority Emulator

AEMA is a prototype that is yet to be automated for a few exotic architectures. For example, Xtensa and PowerPC architecture support automated emulation generally. However, ColdFire and RISC-V architecture must manually configure the fuzzing start address, data register, and memory layout for each program. In future works, we will apply the function of extracting the address at which the data register is first used and memory layout through binary analysis tools or heuristic methods for a fully automated emulation.

6 Related Works

In this section, we describe the features and limitations of previous IoT testing studies, as well as improvements in AEMA. We compared them with AEMA in Table 5 to demonstrate that AEMA complemented emulation availability and firmware compatibility.

Table 5: Comparison of related works and AEMA

System	Architecture compatibility	Emulation without real devices	Automated configuration	Emulation method
FIRMCORN [4]	x86(Little Endian), ARM, MIPS	No	No	CPU emulation with unicorn engines
Pretender [50]	ARM	No	Partial	Machine learning-based modeling
P2IM [5]	ARM	Yes	No	Pattern-based modeling
Firm-AFL [3]	MIPS, ARM	Yes	Partial	Augmented process emulation
μEmu [6]	ARM	Yes	No	Invalidity-guided knowledge inference via symbolic execution

(Continued)

Table 5: Continued

System	Architecture compatibility	Emulation without real devices	Automated configuration	Emulation method
Jetset [7]	x86, ARM, ColdFire	Yes	No	Guided symbolic execution for targeted firmware emulation
AEMA	x86(Little Endian), ARM, MIPS, Xtensa, PowerPC, RISC-V, ColdFire	Yes	Partial	Arbitrated process emulation with keyword extraction technique

6.1 Hardware in the Loop Emulation Frameworks

FIRMCORN [4] is an IoT fuzzing framework that focuses on CPU emulation using unicorn engines to improve the fuzzing throughput and instability of emulation in IoT emulation environments. FIRMCORN [4] improved vulnerability detection using a vulnerable code search algorithm that allowed fuzzers to quickly access suspected vulnerabilities. However, dumping the context information from actual devices is necessary because the initial emulation is configured only by knowing the context information of the firmware. By contrast, AEMA can be operated with only firmware binaries.

Pretender [50] is a system that models peripheral hardware in firmware through machine learning algorithms to increase emulation compatibility. Pretender [50] records the interrupts and memory-mapped inputs/outputs (MMIO) from actual devices and learns them. Subsequently, it models peripherals through a learned MMIO model when the same firmware enters without real-world devices. However, learned models become impractical when emulating unlearned firmware and require actual devices for learning. By contrast, AEMA need not be connected to real devices and is highly compatible with new firmware.

6.2 Hardwareless Emulation Frameworks

P2IM [5] is an emulation framework aimed at bare-metal firmware fuzzing. P2IM [5] abstracts and models APIs between peripherals and processors to mitigate hardware dependency and low scalability. It models registers by identifying and classifying access patterns of registers on peripherals but manually configuring registers and interrupts in the phase of abstracting the model. By contrast, arbitration techniques are used automatically in AEMA to minimize manual configuration before running.

Firm-AFL [3] is the first firmware fuzzing framework to introduce augmented process emulation. Firm-AFL [3] has come up with a compromise between the fuzzing throughput and emulation reliability. It generally operates fuzzing in light user-mode emulation, and full-system emulation is only operated for system calls that user-mode emulation cannot process. However, in Firm-AFL [3], the references of structured inputs are not considered for increasing the path coverage. By contrast, AEMA improved the path coverage by referring to the program path values using the keyword extraction technique through concolic execution.

Firmfuzz [8] is an automated hardware-independent emulation and fuzzing framework. It communicates between the emulator and fuzzer through a web interface. The fuzzer syntactically generates legal input through static analysis and monitors the firmware runtime. However, Firmfuzz [8] is a generation-based fuzzer, so it is less compatible and relatively unlikely to detect vulnerabilities. By contrast, AEMA is relatively likely to detect vulnerabilities because it is a mutation-based fuzzer

capable of generating more unconstrained inputs than a generation-based fuzzer. In addition, keyword extraction techniques were applied to partially match the format so that deep paths could be explored.

Snipuzz [51] is a black-box fuzzing framework for IoT firmware and uses a message snippet inference algorithm. Snipuzz [51] checks the code coverage based on the response message according to the input. This is based on the premise that if the input has accessed a new code, then the response will be different, and thus, the new response will represent an increase in the code coverage. Message snippet inference algorithms are used to identify the input format of firmware through responses. The role of each byte in the message is also inferred by sending an input set that deletes each byte in the initial response message. The roles of each identified byte are grouped to specify the input format and used for the mutation of fuzzing. Snipuzz [51] does not need to know the input format of the target in advance and is relatively less affected by the diversity of firmware. However, because it is black-box testing, code coverage measurements are incomplete such as making the same response even if a new path is covered. In addition, system performance is dependent on the amount of information in the response message. By contrast, because AEMA uses grey-box testing, coverage reliability is not affected by the target and duplicated feedback.

μEmu [6] is an automatic firmware emulation framework that operates through the inference of peripherals using symbolic execution. μEmu [6] works on the principle that complete emulation of peripherals is very challenging and not actually important in firmware analysis. Therefore, it is more useful than peripheral modeling to deliver an appropriate value through symbolic execution whenever each peripheral is accessed. μEmu [6] captures the static and dynamic behaviors of peripherals as symbols during firmware operation, resolves them through symbolic execution, and stores them as a knowledge base. It then works by matching stored caching values when accessing peripheral devices using four hierarchical rules and knowledge-based caching strategies. μEmu [6] has improved emulation success rates and path coverage compared to that reported in previous studies, indicating that register values can be inferred with high accuracy using symbol execution. However, μEmu is only compatible with ARM microcontroller unit (MCU) firmware. By contrast, AEMA can support various architectures other than ARM.

Jetset [7] is a targeted firmware emulation framework that uses symbolic execution. Jetset [7] aims not to explore the entire firmware but to successfully reach the target address so that the code after the target address can be tested and analyzed. Jetset [7] uses symbolic execution to infer the peripheral values encoded inside the firmware because it has to go through the peripheral initialization step after booting to reach the target address. In addition, it selects a specific execution path using the modified Tabu search algorithm and minimizes the distance to the target address. Jetset [7] preferentially searches for the shortest path to reach the target address with the Tabu search algorithm. It backtracks when the current state is matched to the Tabu list, which is an invalid state list. Jetset specifies the invalid state as system reset trigger functions and infinite-loop functions. Jetset [7] improved the emulation stability and alleviated the problem of path explosion in symbolic execution through targeted rehosting. However, Jetset [7] requires manual efforts to identify and configure a memory layout, entry point address, and target address. By contrast, AEMA generally performs an automatic configuration for supportable architectures except for RISC-V and ColdFire.

7 Conclusion

In this paper, we proposed AEMA, an adaptive emulation framework for multi-architecture in IoT firmware testing. To address issues related to compatibility and efficiency that exist in previous firmware testing, we implemented a series of improved techniques, which are as follows: *the minority*

emulator that supports automated emulation for exotic firmware architectures such as Xtensa, PowerPC, and RISC-V; *arbitrated process emulation* that increases the success rate of emulation for major firmware, such as MIPS and ARM, and ensures interconnection between the fuzzer and emulator; and *a keyword extraction technique* that provides references to generate test cases for IoT programs effectively. We evaluated the performance of our system with other IoT testing frameworks and proved the effectiveness of our approaches. The results revealed that AEMA could emulate four exotic firmware architectures and achieve high emulation success rates compared with that reported in previous studies. In addition, AEMA can detect vulnerabilities faster and accomplish more path coverage as compared to other frameworks. Finally, we found a 0-day vulnerability in real-world IoT devices.

Funding Statement: This work was supported by the Ministry of Science and ICT (MSIT), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2022-2018-0-01423) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP), and by MSIT, Korea under the ITRC support program (IITP-2021-2020-0-01602) supervised by the IITP.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] Grand View Research, "Industrial internet of things market size, share trends analysis report by component, by end-use (manufacturing, energy power, oil gas, healthcare, logistics transport, agriculture), and segment forecasts, 2019–2025," 2019. [Online.] Available: https://www.grandviewresearch.com/industry-analysis/industrial-internet-of-things-iiot-market. (Accessed on June. 7, 2021).

[2] Pulse, "Some 100 IoT devices found to be malware infected in Korea," 2022. [Online.] Available: https://pulsenews.co.kr/view.php?year=2022&no=61009. (Accessed on May. 16, 2022).

[3] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu *et al.,* "{FIRM-AFL}: High-throughput greybox fuzzing of IoT firmware via augmented process emulation," in *28th USENIX Security Symp. (USENIX Security 19)*, Santa Clara, CA, USA, pp. 1099–1114, 2019.

[4] Z. Gui, H. Shu, F. Kang and X. Xiong, "Firmcorn: Vulnerability-oriented fuzzing of IoT firmware via optimized virtual execution," *IEEE Access*, vol. 8, pp. 29826–29841, 2020.

[5] B. Feng, A. Mera and L. Lu, "{P2IM}: Scalable and Hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symp. (USENIX Security 20)*, Boston, MA, USA, pp. 1237–1254, 2020.

[6] W. Zhou, L. Guan, P. Liu and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symp.*, Vancouver, B.C., Canada, pp. 2007–2024, 2021.

[7] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Chefckoway *et al.,* "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symp. (USENIX Security 21)*, Vancouver, B.C., Canada, pp. 321–338, 2021.

[8] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe *et al.,* "Firmfuzz: Automated IoT firmware introspection and analysis," in *Proc. of the 2nd Int. ACM Workshop on Security and Privacy for the Internet-of-Things*, London, UK, pp. 15–21, 2019.

[9] J. Kim, J. Yu, H. Kim, F. Rustamov and J. Yun, "FIRM-COV: High-coverage greybox fuzzing for IoT firmware via optimized process emulation," *IEEE Access*, vol. 9, pp. 101627–101642, 2021.

[10] Threadx, "An advanced real-time operating system (RTOS) designed specifically for deeply embedded applications," 2022. [Online.] Available: https://github.com/azure-rtos/threadx/. (Accessed on Oct. 14, 2022).

[11] RTEMS, "Realtime SMP Kernel, networking, file-systems, drivers, BSPs, samples, and testsuite," 2022. [Online.] Available: https://github.com/RTEMS (Accessed on Oct. 14, 2022).
[12] F. Bellard, "QEMU, a fast and portable dynamic translator," *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46, pp. 10–5555, 2005.
[13] D. D. Chen, M. Woo, D. Brumley and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. NDSS*, San Diego, CA, USA, vol. 1, pp. 1, 2016.
[14] 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*, San Diego, CA, USA, 2018.
[15] C. Wright, W. A. Moeglein, S. Bagchi and M. Kulkarni, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–36, 2021.
[16] P. Hambarde, R. Varma and S. Jha, "The survey of real time operating system: RTOS," in *2014 Int. Conf. on Electronic Systems, Signal Processing and Computing Technologies*, Nagpur, India, IEEE, pp. 34–39, 2014.
[17] Y. Zhao and D. Sanán, "Rely-guarantee reasoning about concurrent memory management in zephyr RTOS," in *Int. Conf. on Computer Aided Verification*, Cham, New York, NY, USA, Springer, pp. 515–533, 2019.
[18] B. Ip, "Performance analysis of VxWorks and RTLinux," Technical report, New York, USA, [Online.] Available: http://www.cs.columbia.edu/~sedwards/classes/2001/w4995-02/reports/ip.pdf. (Accessed on Oct. 7, 2022).
[19] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz *et al.,* "{HALucinator}: Firmware Rehosting through abstraction layer emulation," in *29th USENIX Security Symp. (USENIX Security 20)*, Boston, MA, USA, pp. 1201–1218, 2020.
[20] N. A. Quynh and D. H. Vu, "Unicorn: Next generation CPU emulator framework," in *Proc. of the 2015 BlackHat USA conf.*, Las Vegas, NV, USA, 2015.
[21] H. J. Abdelnur, R. State and O. Festor, "KiF: A stateful SIP fuzzer," in *Proc. of the 1st Int. Conf. on Principles, Systems and Applications of IP Telecommunication*, New York, USA, pp. 47–56, 2007.
[22] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Int. Conf. on Security and Privacy in Communication Systems*, Cham, Dallas, TX, USA, Springer, pp. 330–347, 2015.
[23] Boofuzz, "A fork and successor of the Sulley Fuzzing Framework," 2022. [Online.] Available: https://github.com/jtpereyda/boofuzz. (Accessed on Oct. 22, 2022).
[24] D. Yang, Y. Zhang and Q. Liu, "Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs," in *2012 IEEE 11th Int. Conf. on Trust, Security and Privacy in Computing and Communications*, Liverpool, UK, IEEE, pp. 1070–1076, 2012.
[25] P. Godefroid, M. Y. Levin and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.
[26] K. M. Alshmrany, R. S. Menezes, M. R. Gadelha and L. C. Cordeiro, "FuSeBMC: A white-box fuzzer for finding security vulnerabilities in C programs (competition contribution)," in *Fundamental Approaches to Software Engineering: 24th International Conference, FASE 2021,* March 27–April 1, 2021, Proceedings 24, Luxembourg City, Luxembourg: Springer International Publishing, vol. 12649, pp. 363, 2021.
[27] H. Peng, Y. Shoshitaishvili and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *2018 IEEE Symp. on Security and Privacy (SP)*, San Francisco, CA, USA, IEEE, pp. 697–710, 2018.
[28] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang *et al.,* "Driller: Augmenting fuzzing through selective symbolic execution," *Proc. NDSS*, vol. 16, no. , pp. 1–16, 2016.
[29] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," in *Proc. of the 2019 ACM SIGSAC Conf. on Computer and Communications Security*, London, UK, pp. 2637–2639, 2019.
[30] AFL, "American fuzzy lop," 2015. [Online.] Available: http://lcamtuf.coredump.cx/afl/. (Accessed on Nov. 11, 2021).
[31] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symp. on Security and Privacy (SP)*, San Francisco, CA, USA, IEEE, pp. 711–725, 2018.

[32] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie *et al.,* "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security*, Toronto, Canada, pp. 2095–2108, 2018.
[33] J. Wang, B. Chen, L. Wei and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st Int. Conf. on Software Engineering (ICSE)*, Montreal, QC, Canada, IEEE, pp. 724–735, 2019.
[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida *et al.,* "VUzzer: Application-aware evolutionary fuzzing," *Proc. NDSS*, vol. 17, pp. 1–14, 2017.
[35] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed automated random testing," in *Proc. of the 2005 ACM SIGPLAN conf. on Programming Language Design and Implementation*, Chicago, Illinois, USA, pp. 213–223, 2005.
[36] K. Sen, D. Marinov and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.
[37] C. Cadar, D. Dunbar and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, SanDiego, CA, USA, pp. 209–224, 2008.
[38] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino *et al.,* "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symp. on Security and Privacy (SP)*, San Jose, CA, USA, IEEE, pp. 138–157, 2016.
[39] F. Saudel and J. Salwan, "Triton: Concolic execution framework," in *Symp. sur la sécurité des technologies de l'information et des communications (SSTIC)*, Rennes, France, 2015.
[40] I. Yun, S. Lee, M. Xu, Y. Jang and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symp. (USENIX Security 18)*, Baltimore, MD, USA, pp. 745–761, 2018.
[41] Binwalk, "Firmware analysis tool," 2022. [Online.] Available: https://github.com/ReFirmLabs/binwalk. (Accessed on Oct. 14, 2022).
[42] R. Rohleder, "Hands-on ghidra-a tutorial about the software reverse engineering framework," in *Proc. of the 3rd ACM Workshop on Software Protection*, London, UK, pp. 77–78, 2019.
[43] Firmware-mod-kit, "A collection of scripts and utilities to extract and rebuild Linux based firmware images," 2022. [Online.] Available: https://github.com/rampageX/firmware-mod-kit. (Accessed on May. 22, 2022).
[44] A. Mera, B. Feng, L. Lu and E. Kirda, "DICE: Automatic emulation of dma input channels for dynamic firmware analysis," in *2021 IEEE Symp. on Security and Privacy (SP)*, San Francisco, CA, USA, IEEE, pp. 1938–1954, 2021.
[45] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang *et al.,* "Firmae: Towards large-scale emulation of IoT firmware for dynamic analysis," in *Annual Computer Security Applications Conf.*, Austin, USA, pp. 733–745, 2020.
[46] M. Böhme, V. T. Pham and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security*, Vienna, Austria, pp. 1032–1043, 2016.
[47] S. Bugrara and D. Engler, "Redundant state detection for dynamic symbolic execution," in *2013 USENIX Annual Technical Conf. (USENIX ATC 13)*, San Jose, CA, USA, pp. 199–211, 2013.
[48] V. Chipounov, V. Georgescu, C. Zamfir and G. Candea, "Selective symbolic execution," in *Proc. of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, Lisbon, Portugal, CONF, 2009.
[49] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
[50] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry *et al.,* "Toward the analysis of embedded firmware through automated re-hosting," in *22nd Int. Symp. on Research in Attacks, Intrusions and Defenses (RAID)*, Beijing, China, pp. 135–150, 2019.
[51] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen *et al.,* "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," in *Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security*, pp. 337–350, 2021. https://doi.org/10.1145/3460120.3484543