FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation

Yaowen Zheng[1,2,3]*, *Ali Davanian*[2], *Heng Yin*[2], *Chengyu Song*[2], *Hongsong Zhu*[1,3], and *Limin Sun*[1,3]†
[1] Beijing Key Laboratory of IoT Information Security Technology,
Institute of Information Engineering, CAS, China
[2] University of California, Riverside, USA
[3] School of Cyber Security, University of Chinese Academy of Sciences, China
{zhengyaowen,zhuhongsong,sunlimin}@iie.ac.cn, adava003@ucr.edu, {heng,csong}@cs.ucr.edu

Abstract

Cyber attacks against IoT devices are a severe threat. These attacks exploit software vulnerabilities in IoT firmware. Fuzzing is an effective software testing technique for vulnerability discovery. In this work, we present FIRM-AFL, the first high-throughput greybox fuzzer for IoT firmware. FIRM-AFL addresses two fundamental problems in IoT fuzzing. First, it addresses compatibility issues by enabling fuzzing for POSIX-compatible firmware that can be emulated in a system emulator. Second, it addresses the performance bottleneck caused by system-mode emulation with a novel technique called augmented process emulation. By combining system-mode emulation and user-mode emulation in a novel way, augmented process emulation provides high compatibility as system-mode emulation and high throughput as user-mode emulation. Our evaluation results show that (1) FIRM-AFL is fully functional and capable of finding real-world vulnerabilities in IoT programs; (2) the throughput of FIRM-AFL is on average 8.2 times higher than system-mode emulation based fuzzing; and (3) FIRM-AFL is able to find 1-day vulnerabilities much faster than system-mode emulation based fuzzing, and is able to find 0-day vulnerabilities.

1 Introduction

The security impact of IoT devices on our life is tremendous. By 2020, the number of connected IoT devices will exceed the number of people [10]. This creates an unprecedented attack surface leaving almost everybody at danger. Even currently, the hackers leverage the lack of security in IoT devices to create large botnets (e.g., Mirai, VPNFilter and Prowli). These malware attacks exploit the vulnerabilities in IoT firmware to penetrate into the IoT devices. As a result, it is crucial for defenders to discover vulnerabilities in IoT firmware and fix them before attackers.

Fuzzing, a software testing technique that feeds a program with random inputs, has approved to be very effective in finding vulnerabilities in real-world programs. In particular, AFL [34], a coverage-guided greybox fuzzing tool, has been used widely in both industry and academia. For instance, most of the finalists in DARPA Cyber Grand Challenge used AFL as the primary vulnerability discovery component [2].

Challenges in IoT firmware fuzzing. Despite the effectiveness of fuzzing for programs on general-purpose platforms, it is generally not feasible to directly apply fuzzing on IoT firmware, due to its strong dependency on the actual hardware configuration. For instance, simply extracting a user-level program from a Linux-based firmware and fuzzing this program with AFL would not work in most cases.

To this end, recent researches propose a series of solutions, ranging from directly fuzzing the IoT devices (e.g., IoTFuzzer [14]), a hybrid solution that combines hardware and software emulation (e.g., AVATAR [33]), to a full system emulation (e.g., Firmadyne [13]). As a recent study by Muench et al. [28] points out, full system emulation yields the highest throughput, because IoT devices are much slower than a desktop workstation or a server.

Throughput is a key factor for the effectiveness of fuzzing. However, even for full system emulation, its performance is far from being ideal. According to our evaluation (§5), full system emulation is about 10 times slower than user-mode emulation (which is used by AFL). 10 times slowdown means approximately 10 times more computing resources are needed to find a vulnerability in an IoT program than its desktop counterpart. According to our analysis (§2.4), part of the enormous runtime overhead of full-system emulation comes from software implementation of memory management unit (i.e., SoftMMU) that is used to translate a guest virtual address into a host virtual address for every single memory access happening in the virtual machine. The other part of the overhead comes from the system calls emulation overhead.

*The work was done while visiting University of California, Riverside
†Corresponding author

Our solution: greybox fuzzing via augmented process emulation. In this work, we present, to the best of our knowledge, the first greybox fuzzer for IoT firmware, that achieves two design goals simultaneously: (1) *transparency* that is no modification should be needed for the program in firmware to be fuzzed, and (2) *efficiency* that is the fuzzing throughput of the overall system should come close to that of the user-mode emulation. Our key insight is to find a novel combination of full-system emulation and user-mode emulation to achieve the best of two worlds: generality from full-system emulation and efficiency from user-mode emulation.

More specifically, we propose a new technique called "augmented process emulation". As the name suggests, its main idea is to augment process (or user-mode) emulation with full system emulation. The program to be fuzzed is mainly run in user-mode emulation to achieve high efficiency, and switches to full system emulation only when necessary to ensure correct program execution, thus achieving generality.

To evaluate the feasibility of this technique, we implement a prototype system called FIRM-AFL, on top of AFL [34] and Firmadyne [13]. From a user's perspective, using FIRM-AFL, we can conduct coverage-guided greybox fuzzing on a user-specified program from an IoT firmware, the same as fuzzing a normal user-level program using AFL. Under the hood, FIRM-AFL occasionally switches to the full system emulation mode in Firmadyne, to ensure the given program can be correctly emulated.

We have evaluated FIRM-AFL with standard benchmarks and a set of real-world IoT firmware images. The evaluation results showed that (1) FIRM-AFL can faithfully emulate the target programs as if they were running in full-system emulation; (2) compared to a full-system emulation based fuzzer (TriforceAFL [29] with lightweight snapshot enabled), the throughput of FIRM-AFL is 8.2 times higher on average and (3) FIRM-AFL can find 1-day vulnerabilities 3 to 13 times faster than full-system emulation based fuzzer, and was able to find two 0-day vulnerabilities within 8 hours on a single machine.

Contributions. In summary, we make the following contributions in this paper:

- We point out that full system emulation exerts significant runtime overhead, and is far from ideal to serve as the base for IoT firmware fuzzing. We further investigate the root cause of this runtime overhead.
- We propose a novel technique called "augmented process emulation", to reconcile the contradictory characteristics of full-system emulation (high generality and low efficiency) and user-mode emulation (low generality and high efficiency).
- We design and implement the first coverage-guided greybox fuzzing platform for IoT firmware, FIRM-AFL.
- We extensively evaluate our system and show the overhead for each part of our system. Our improvements lead to 8.2 times speedup on average. As a result, FIRM-AFL could find 1-day vulnerabilities 3 to 13 times faster than full-system emulation, and was able to find two new vulnerabilities within 8 hours on a single machine.
- The current implementation of FIRM-AFL supports three CPU architectures, including mipsel, mipseb and armel, which cover 90.2% firmware images in the Firmadyne datasheet [4]. The source code of FIRM-AFL can be found at `https://github.com/zyw-200/FirmAFL`.

2 Background and Motivation

2.1 Fuzzing

Fuzzing is a software testing technique that aims to find bugs by executing the target program with random inputs and looking for interesting program behaviors such as the crashes. Based on how much information is collected and used from the execution, fuzzers can be categorized into blackbox, whitebox and greybox. A blackbox fuzzer treats the target program as a blackbox and does not utilize any feedback from the execution to guide the generation of random inputs. This approach was originally used to test Linux utilities [26]. On the other hand, a whitebox fuzzer selects the inputs based on a deep insight into the target program. This is usually achieved through expensive program analysis techniques like dynamic taint analysis and symbolic execution [22]. Finally, a greybox fuzzer improves the testing by utilizing limited information collected with lightweight monitoring techniques (e.g., code coverage).

The most popular greybox fuzzers are coverage-guided fuzzers. These fuzzers instrument the target program to collect code coverage information. The collected information is then used to guide the input generation—inputs that explore new execution paths will be used as *seeds* to generate new inputs while inputs that did not yield new coverage will be discarded. This simple strategy is extremely effective in practice. In fact, greybox fuzzers can even outperform whitebox fuzzers when targeting real-world applications. Their secret is *speed*, the lightweight instrumentation allows greybox fuzzers to execute hundreds or thousands times more inputs than whitebox fuzzers [32]. In other words, throughput is paramount for greybox fuzzers.

AFL [34] is a well-known greybox fuzzer. It can instrument the program either statically or dynamically. Static instrumentation is preferred when the source code is available. When the source code is not available, e.g., when fuzzing commercial off-the-shelf (COTS) programs, AFL utilizes a binary translator (i.e., user-mode emulation provided by QEMU [12]) to perform the instrumentation. For most IoT devices, because source code and design documents are often proprietary and

only firmware image might be available, dynamic instrumentation is the only viable option. As a matter of fact, even extracting the binary from the firmware is not always straightforward [14].

2.2 QEMU

QEMU [12] is a fast processor emulator based on dynamic binary translation. Unlike traditional emulators that interpret the target program instruction-by-instruction, QEMU translates several basic blocks at a time. More importantly, it caches translated blocks and uses block chaining to link them together. This allows the execution to remain inside the code cache (i.e., the logic of the target program) for the most of the time thus minimizes the overhead of the translation. Dynamic instrumentation can be performed during the translation to introduce new functionalities, such as branch monitoring [34] and taint propagation [19, 23].

Besides the translation of instructions, the next most important task is address space translation. The translation is done very differently based on the execution mode. In system mode, QEMU implements a software Memory Management Unit (MMU) to handle memory accesses. The software MMU maps Guest Virtual Addresses (GVAs) to the Host Virtual Addresses (HVA). This mapping process is transparent to the guest operating system (OS) meaning that QEMU still allows the guest OS to set up the GVA to Guest Physical Address (GPA) mapping through the interface of page tables and to handle page faults. Under the hood, QEMU inserts a GVA to GPA translation logic for every memory access. To speed up the translation, QEMU uses a software Translation Lookaside Buffer (TLB) to cache the translation results. Moreover, to avoid invalidating the code cache and block chaining whenever the address translation changes, all translated blocks are indexed using GPA and the block chaining is only performed when the two basic blocks are within the same physical page. GPA to HVA mapping is done using a linear mapping (i.e., HVA = GPA + OFFSET).

In contrast to system-mode emulation, in user-mode emulation, the Host Virtual Address (HVA) is calculated as the Guest Virtual Address (GVA) plus a constant offset. So this translation is much faster than the one in system-mode emulation.

2.3 Testing IoT Firmware

As IoT devices become a popular attack target, testing IoT programs to find vulnerabilities also becomes important. There are two main challenges in testing IoT programs. The first challenge is compatibility: many IoT programs depend on special hardware components of the device thus cannot be tested without proper support. The second challenge is code coverage: blackbox fuzzers are known to have low code coverage while whitebox fuzzers cannot scale to slightly larger code base [20]. Table 1 compares some representative efforts on IoT firmware testing using these two metrics.

Avatar [33] aims to enable dynamic program analysis for embedded firmware by providing better hardware component support. It achieves this goal through constructing a hybrid execution environment consists of both a processor emulator (QEMU) and real hardware where Avatar acts as a software proxy between the emulator and the real hardware. This allows Avatar to utilize the emulator to execute and analysis the instructions while channeling the I/O operations to the physical hardware. As a demonstration, the authors have applied S2E [15], a whitebox fuzzing tool to find vulnerabilities in the Redwire Econotag Zigbee sensor. Due to the involvement of whitebox fuzzing and slow hardware, the throughput of Avatar is expected to be very low.

IoTFuzzer [14] performs blackbox fuzzing directly on the real device. Its main advantage over previous blackbox fuzzing based approaches is that it performs the fuzzing through the companion mobile app of the target device. By automatically analyzing the data flow in the companion app to better understand the communication protocol, IoTFuzzer can generate better test cases that are more likely to trigger a bug. That said, based on its evaluation, IoTFuzzer never exceeds a throughput of 1 test case per second, which is slow (based on Table III in [14]).

Although it does not perform fuzzing, Firmadyne [13] adds hardware support for IoT firmware to the system mode QEMU. It provides support for both ARM and MIPS architectures that are popular among the IoT manufacturers. For hardware support, Firmadyne fully emulates the system by modifying the kernel and drivers to handle the IoT exceptions due to the lack of actual hardware. Compared to the former two solutions, this solution is easier to adapt to new IoT firmware and programs. The throughput of full-system emulation is usually better than the native execution [28].

Muench et al. [28] compare the throughput of a blackbox fuzzer [24] under different configurations, including native execution (directly sending inputs to the hardware), partial emulation (redirecting only hardware requests to the hardware), and full emulation. Their emulation is based on image replaying capability provided by PANDA [19]. They concluded that full emulation (FE) has the highest throughput mainly because the IoT processors are much slower than desktop processors. However, even in the best case, the throughput did not exceed 15 test cases per second [1].

AFL [34] is a well-known greybox fuzzer that can support binary-only fuzzing through user-mode QEMU. Unfortunately, lacking special hardware support, user-mode QEMU can not successfully emulate most IoT programs. For example, AFL with user-mode QEMU failed on all the programs used in our evaluation (Table 3). Moreover, simply adopting a full system emulator (e.g., Firmadyne) does not fully solve

[1] They reported 53390 cases/hour which is equal to 15 cases/second

	Avatar [33]	IoTFuzzer [14]	Firmadyne [13]	Muench et al. [28]	AFL [34]
Technique	Whitebox fuzzing	Blackbox fuzzing	PoC	Blackbox fuzzing	Greybox fuzzing
Compatibility	High	High	High	High	Low
Hardware Support	Hybrid	Real	Emulation	Mixed	None
Code Coverage	Medium	Low	N/A	Low	High
Throughput	Very Low	Low	Medium	Low to Medium	High
Zero-day Detection	Yes	Yes	No	Yes	Yes

Table 1: Comparison of IoT firmware testing tools.

the problem because the throughput is low.

In summary, existing IoT firmware testing tools do not provide satisfying code coverage yet sate-of-the-art fuzzers (e.g., AFL) cannot be easily applied to test IoT programs. So far, there is no greybox IoT fuzzer, not to mention a greybox IoT fuzzer with good throughput.

2.4 Motivations

Given the unsatisfying status-quo of IoT firmware testing tools, we aim to enable high-throughput greybox fuzzing for IoT programs. To this end, we decide to build the fuzzer based on emulation. This choice is based on two reasons. First, greybox fuzzing requires collecting execution information (e.g., branch coverage) to guide test case generation. As mentioned in §2.1, this is usually done through lightweight instrumentation. Since most IoT programs are only distributed in binary format, emulator-based instrumentation is the best available option. The second reason is performance. Although it might be possible to run instrumented binaries directly on the device, Muench et al. [28] have shown that full-emulation-based approach is actually faster than the real device, because the desktop processors are much faster.

Unfortunately, simply adopting a full system emulator (e.g., Firmadyne [13]) does not fully solve the problem because the throughput is not enough. For example, even with the full-emulation configuration, the fuzzer used in [28] never exceeded 15 test cases per second. To understand the bottleneck, we profiled the execution time of two networking tools (`basename` and `uptime`) under full-system emulation (with lightweight snapshot) and user-mode emulation. The results are shown in Table 2. Based on this measurement, we can see that the throughput of fuzzing can be significantly boosted if we can apply user-mode emulation to the target program. There are several bottlenecks that contribute to the execution time difference.

- *B1. Memory address translation*. In full-system emulation, QEMU uses a software MMU to perform address translation for *every* memory access. In contrast, in user-mode emulation, the address translation is much simpler. So even if we just consider time spent in user-mode execution, user-mode emulation uses much less time.
- *B2. Dynamic code translation*. The code translation process in user-mode emulation is faster than the full-system mode. In full-system mode, block chaining is limited to basic blocks in the same physical page, which means the translator is invoked more often than in user-mode emulation.
- *B3. Syscall emulation*. In user-mode emulation, system calls are handled directly by the host OS and hardware. Therefore, it is significantly faster than full-system emulation where the OS also runs in the emulator and the hardware devices are also emulated. Although hardware emulation is necessary to allow the target program to run correctly, not all system calls would rely on the special hardware. In other words, not all system calls require emulation.

In this work, we address all three bottlenecks to improve the throughput of IoT program fuzzing.

3 Augmented Process Emulation

3.1 Overview

The goal of this work is to enable high-throughput greybox fuzzing for IoT programs. As discussed in §2, to achieve this goal, we need to overcome two challenges: compatibility and performance. The first challenge can be solved through full-system emulation but this would result in poor performance. The second challenge can be solved through user-mode emulation but would result in poor compatibility. In this section, we present *augmented process emulation*, a new approach that brings the best of both full-system emulation and user-mode emulation.

Problem statement. Generally speaking, the goal of augmented process emulation is to correctly execute a program of an IoT firmware in a user-mode emulator, given the following requirements are satisfied:

(1) The firmware can be correctly emulated in a system emulator (e.g., system-mode QEMU). Fortunately, with the help of Firmadyne [13], a large portion of IoT firmware images are able to meet this requirement.

	system mode (ms)					user mode (ms)			
program	overall	sys exec	sys code trans	user exec	user code trans	overall	sys exec	user exec	user code trans
`basename`	4.08	1.79	0.53	1.41	0.35	0.34	0.02	0.11	0.22
`uptime`	7.48	2.39	0.76	2.79	1.55	0.89	0.04	0.31	0.54

Table 2: Runtime performance of system mode and user mode emulation

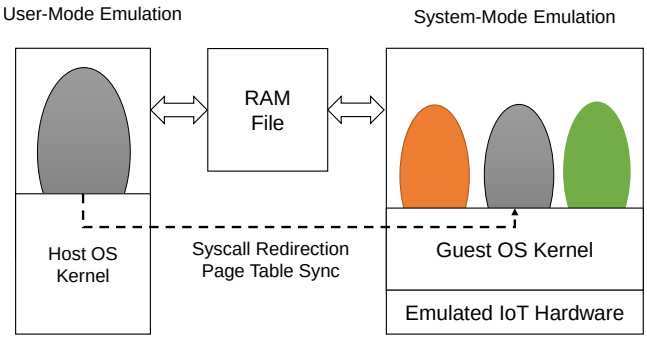

Figure 1: Overview of Augmented Process Emulation

(2) The firmware runs a POSIX-compatible operating system (OS). Fortunately, many IoT firmware images use Linux as the OS hence satisfy this requirement.

With augmented process emulation, we aim to achieve the following design goals:

- *Transparency.* The user-level program running in the augmented process emulation should behave as if it were run in the system-mode emulation.
- *High efficiency.* Since throughput is a dominating factor for fuzzing, the augmented process emulation needs to be as efficient as possible. Ideally, it should approximate the performance of pure user-mode emulation.

Solution overview. To achieve the design goals mentioned above, we resort to combine user-mode emulation with system-mode emulation in a novel manner. Figure 1 illustrates the overview of our solution.

At first, the IoT firmware boots up in the system-mode emulator and the user-level programs (including the one to be fuzzed) are launched properly inside the emulator. After the program to be fuzzed has reached at a predetermined point (e.g., the entry point of main function, or after receiving the first network packet), the process execution is then migrated to the user-mode emulation in order to gain high execution speed. Only at rare occasions, the execution is migrated back to the system-mode execution to ensure the correctness of execution.

To minimize the migration cost, the memory state is shared between these two emulation modes. More concretely, the physical memory of the virtual machine for the system-mode emulation is allocated as a memory-mapped file, called RAM file. This RAM file is also mapped into the address space of the user-mode emulation. Note that system-mode emulation and user-mode emulation access this RAM file in different ways. System-mode emulation treats the RAM file as physical memory, and thus accesses it by physical address, while user-mode emulation accesses the shared memory by virtual address. Therefore, the physical pages in the RAM file need to be mapped into the address space of user-mode emulation by their virtual addresses at a page granularity. As a result, when a page mapping is not established in the user-mode emulation, the process execution needs to be migrated to the system-mode emulation to establish this mapping. We will discuss more details about the memory mapping in §3.2.

With a proper memory mapping, the process should be able to execute correctly in the user-mode emulation, until it reaches a system call. Directly executing the system call locally on the host OS would not work in general, because the host OS and the OS in IoT firmware are different and the underneath hardware layers are also different. To ensure transparency, we need to migrate the execution to the system-mode emulation to process this system call. When the system call returns, we migrate the execution back to the user-mode emulation. More details will be discussed in §3.3.

3.2 Memory Mapping

Bootstrapping. When fuzzing a program with AFL, the program executes to a predetermined point, and then the fork server of AFL will repeatedly fork a new program instance on this point (which is referred to as fork point) and feed random inputs. Similarly, in this setting, we will boot up the IoT firmware in system-mode emulation and further launch the specified IoT program. Using Virtual Machine Introspection (VMI) provided by DECAF [23] (a system emulation based dynamic analysis platform), we are able to monitor the execution of the specified IoT program and get notified when the execution reaches to the predetermined fork point.

At this moment, we will walk the page table of the specified process and collect the virtual to physical page mapping information and send it over to the user-mode emulation side. Then for each mapping of virtual address (`va`) to physical address (`pa`), the user-mode emulation side establishes a mapping by calling `mmap` as below:

```
mmap(va, 4096, prot, MAP_FILE, ram_fd, pa);
```

The code above is self-explanatory. Essentially, we map a page of the RAM file with the physical address as offset into a specified virtual address. The argument `prot` is determined by the protection bits from the corresponding page table entry.

From this point onward, the execution in system-mode emulation is paused, the CPU state is sent over to user-mode emulation, and the execution resumes there.

Page fault handing. During the process execution in user-mode emulation, if the accessed memory addresses have already been mapped in this address space, the execution should proceed successfully. Otherwise, the host processor will raise a page fault. We register a signal handler for page fault in user-mode emulation, so the host OS will pass along the page fault event to the user-mode emulation. On receiving this signal, the user-mode emulation records the CPU state at the faulting instruction, pauses the execution, and passes the CPU state to the system-mode emulation side, expecting that the page fault can be handled in the system-mode emulation and a new mapping for the faulting virtual address can be established.

When the system-mode emulation receives the CPU state and resumes execution, the emulated processor will raise a page fault, since the page is not present. The page fault handler in the OS of the IoT firmware will respond to this page fault and attempt to establish the mapping. Most likely, this mapping will be established by the OS sooner or later (depending on the scheduling of numerous kernel threads and interrupt handlers) and the instruction that causes the page fault will be re-executed. In very rare cases, if the OS cannot establish a mapping for various reasons, it will kill the process.

A key question here is to determine when the page mapping has been established or an error occurs, so we can switch back to the user-mode emulation to maximize execution speed. The answer to this question is in fact non-trivial, because the OS is handling multiple tasks simultaneously and enormous amount of context switches may happen in the meantime.

To capture the right moment when a mapping is established, we instrument the end of each basic block. If the execution is currently within the specified process (or thread), it means the execution has returned from the kernel to the user space to resume the faulting instruction. The mapping must be present in the software TLB. So we can just directly find the mapping there. At this moment, we pass the mapping information and the CPU state back to the user-mode emulation, which will create this new mapping by calling `mmap` and resume the execution.

If for some reasons, an error occurs and the process gets killed, we can rely on the VMI (Virtual Machine Introspection) capability provided by DECAF [23] to get notified, and then the whole execution on both sides get terminated.

Preload page mapping. Modern operating systems load memory pages in a lazy manner. Although when a new process starts, all code pages are assigned into its address space, a mapping from each virtual page to its physical page is not really established until a page fault caused by the first memory access to it.

This lazy design has adverse effect on fuzzing performance. As we will discuss in §4.1, a child process is repeatedly forked from the parent process for each fuzzing iteration, and thus there are always a series of page faults caused by un-mapped code pages. This is especially harmful for our system, because the overhead of page fault handling is much more expensive than handling it locally on the host OS.

To solve this problem, we decide to preload the code pages of the given process in the physical memory and perform the mapping between the two modes. This helps us avoid repeatedly loading the code pages at every fuzzing iteration, and hence speed up the fuzzing throughput. To do that, we simulate the access to each program code page in the system-mode emulation during the bootstrap, to force the OS to map each page into the process' address space. As a result, we can reduce the number of page faults caused by these pre-loaded pages.

3.3 System Call Redirection

System calls and their implementation in IoT programs are different because of the underlying IoT hardware, firmware and requirements. Consequently, user-mode emulation of an IoT program will likely fail if the exceptions caused by the system calls are not properly handled (see §2). For example, most IoT devices have network interfaces that are not available on a local emulator. When an IoT program in the user-mode emulation executes a system call that needs to interact with a specific network interface in the IoT system, there will be a fault that needs to be handled. Another example is a system call that accesses NVRAM that is undefined for a desktop computer.

Therefore, to ensure execution correctness, we must redirect the system calls from the user-mode emulation to the system-mode emulation. More specifically, when the user-mode emulation encounters a system call, it pauses the execution, saves the current CPU state, and sends it over to the system-mode emulation. The system-mode emulation receives the CPU state and resumes execution. This will cause a mode switch into the kernel mode in the guest system to process the corresponding system call. Again, since the guest OS kernel is multi-tasking, there might be many context switches happening before the system call returns. So similar to how we handle page faults, we will instrument the end of each basic block. If the current basic block is in the kernel space, but next program counter is in the user level, and the current execution context is for the thread that makes the system call, we detect the moment when the system call returns. Then at this moment, we pause the execution in the system-mode emulation, save the CPU state, and pass it back to the user-mode

emulation, which will then resume the execution.

Optimizing filesystem-related system calls. While examining the system calls made by a set of IoT programs, we realize that many system calls are related to the file system. The IoT programs either attempt to access files or directories that already exist in the firmware or are newly created for only temporary uses. We propose an optimization for this set of system calls. We map the file system from the firmware image, and mount it as a directory in the host OS, such that the user-mode emulation can directly access it. In this way, the user-mode emulation can directly pass through the file-system related system calls to the host OS, instead of redirecting them to the system-mode emulation.

As shown in §5.3, filesystem-related system calls take a significant portion among all system calls, and thus this optimization makes a significant contribution for the final performance.

4 Firm-AFL Design and Implementation

Leveraging the technique described in §3, we design and implement FIRM-AFL, an enhancement of AFL [34] for fuzzing IoT firmware. In §4.1, we first describe the workflow of AFL, and then in §4.2, we present how we integrate augmented process emulation into the workflow of AFL.

4.1 Workflow of AFL

AFL is a coverage-guided greybox fuzzer. It maintains a seed queue that stores all the seeds, including the initial seeds chosen by the user as well as the ones that are mutated from the existing seeds and cause the program to reach unique code coverage.

The main program that drives the fuzzing process is `afl-fuzz`. It picks a seed from the seed queue, performs a random mutation, generates an input, and feeds this input to the target program (assuming it is a binary executable).

In order to collect the code coverage information from the execution of the target program, AFL starts the program using the user-mode QEMU, and instruments the branch transitions of the target program, and the code coverage information is encoded and stored in a bitmap.

Since during fuzzing we need to execute the target program repeatedly, AFL utilizes "fork" as a mechanism to speed up this process. It first runs the target program up to a certain point (e.g., the entry point of the main function) such that the program's code and data have been properly initialized, and then repeatedly forks a child process from it. In this way, the initial setup of a new process is skipped. For this reason, the parent process is called `fork-server`. Then the input is fed into the forked child process, and the coverage information is collected and stored in the bitmap, which is shared among

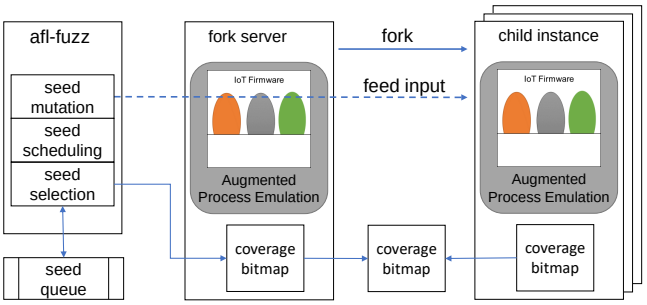

Figure 2: Overview of FIRM-AFL

all three processes (`afl-fuzz`, `fork-server`, and the child instance). `afl-fuzz` will compare the bitmap from the child instance and the accumulative bitmap from all past executions to determine if this mutated input should be kept as a new seed and stored in the seed queue.

4.2 AFL with Augmented Process Emulation

We would like to keep the workflow of AFL intact, but allow AFL to fuzz a target program in an IoT firmware image. To do so, we replace the user-mode QEMU with augmented process emulation, and the rest of the components remain unchanged. The new workflow is illustrated in Figure 2.

Bootstrapping. To fuzz a program in the IoT firmware image, we need to boot up the firmware image and launch the program after the system boots up. This is done in the system-mode emulation within `fork-server`.

We leverage Firmadyne [13] to correctly emulate a firmware image. We further integrate DECAF [23] with Firmadyne to make use of its VMI (Virtual Machine Introspection) capability. In this way, we are able to capture the precise moment when the target program is started or terminated. We can also know when the execution of the target program has reached the pre-determined fork point.

Forking. The default fork point chosen by AFL is the entry point of the main function. In our case, we are interested in finding vulnerabilities in the IoT programs that are triggered through the network interface. Therefore, we hook the network-related system calls. And the first invocation of any of these system calls becomes the fork point.

In the standard workflow of AFL, we can simply leverage the `fork` system call to fork a child process and start the next fuzzing instance. In our case, we not only need to fork a child process for the user-mode emulation, but also "fork" a new virtual machine instance for the system-mode emulation, because two modes must synchronize with each other.

Actually forking a new virtual machine would be too expensive. Instead, we can make a snapshot of the virtual machine

at the fork point, and when one fuzzing execution is finished, we can restore the snapshot. System-mode QEMU offers `save_snapshot` function that saves all the CPU registers and the memory space to a specific file. However, file write/read operations would still be very slow.

In our system, we implement a lightweight snapshot mechanism based on the Copy-on-Write principle. More concretely, we first mark the RAM file mapped into the system-mode QEMU as read-only. Then a memory write will cause a page fault. We make a copy of the page, and then mark this page as write-able. As such, we record all memory pages that have been modified during one fuzzing execution. When restoring the snapshot, we only need to write these recorded pages back.

Feeding input. The inputs are fed through instrumenting system calls. For the IoT programs that are receiving input from network interface, we instrument the network-related system calls in the user-mode emulation directly, so we don't need to redirect these system calls to the system-mode emulation.

Collecting coverage information. Since most of execution happens in the user-mode emulation and system-mode emulation is only needed for handling page faults and some system calls, we can simply instrument the branch transitions in user-mode QEMU to compute the coverage bitmap, just like how the original AFL does it in user-mode QEMU.

5 Evaluation

In this section, we evaluate the prototype implementation of our fuzzer FIRM-AFL. The purpose of this section is to test whether our approach has resolved the performance bottlenecks and achieved the two design goals. In short, we would like to answer following questions:

- *Transparency.* Can FIRM-AFL fuzz programs extracted from IoT firmware as if they are running inside a full-system emulator?
- *High efficiency.* How close is FIRM-AFL's throughput (executions/sec) to the throughput of a pure user-mode emulation based fuzzer?
- *Effectiveness of optimization.* Do our optimization techniques successfully resolved the performance bottlenecks we identified?
- *Effectiveness in vulnerability discovery.* How effective is FIRM-AFL in finding real vulnerabilities in IoT firmware?

Experiments setup. We used three sets of programs in our evaluation. The first set of programs are two standard benchmarks: `nbench` [9] and `lmbench` [7]. They are used to access the correctness of the emulation and the overhead of the emulation. The second set of programs consist of seven IoT programs from four different vendors (Table 3). We selected these program since they are the key service programs that handle network requests thus are good targets for remote attacks. They are used to access the performance of greybox fuzzing. The third dataset is the Firmadyne dataset which includes firmwares whose HTTP and uPnP services are related to 15 1-day exploits (Table 6). We collected them to evaluate the transparency and effectiveness of FIRM-AFL in vulnerability discovery.

Experiments (except the ones in §5.4) are conducted on a 8-core Intel(R) Core(TM) i7-3940XM 3.00GHz CPU machine with 23.5GB of RAM 1TB hard disk . The operating system is Ubuntu 16.04.5 LTS. The version of QEMU and AFL is 2.10.1 and 2.06b. We obtain each measurement value after every ten iterations. Our final reported numbers are the average value of 20 measurements. By default, we set fork point at the position after the network data received, and feed the random input provided by AFL engine.

5.1 Transparency

To evaluate the transparency of our augmented process emulation, we first evaluated our emulator with the `nbench` test suite. After generating the output, the benchmark will compare the outputs with expected outputs. If the generated outputs are wrong, then it implies the emulation is not correct. The results showed that our system can finish all the benchmarks without errors.

We also empirically evaluated the transparency of FIRM-AFL using the Firmadyne dataset [4]. We collected 120 firmware images with HTTP services and unique device models. We first tried to run HTTP service programs in them directly using user-mode QEMU. We extracted the file systems from the firmware images and used `chroot` to mount the file systems. However, all these programs crashed at the very beginning due to the lack of expected system environment. Then we tried to run them with normal inputs (the initial seeds) under full-system emulation, as well as under augmented process emulation. We observed that in both settings, all the programs could run properly. For each program, we further compared the system call sequences generated under full-system emulation as well as augmented process emulation, and confirmed that the system call sequences were identical.

Finally, we evaluated a set of exploits targeting known vulnerabilities listed in Table 6. For each vulnerability, we fed a proof-of-concept (PoC) exploit in both full-system emulation and augmented process emulation and compared the execution traces. We confirmed that the collected two traces are

Program	Size (KB)	Description	Vendor	Devices	Model	Version	CPU Arch
cgibin	129.4	CGI binary program	DLINK	Router	DIR-815	1.01	MIPSEL
httpd	90.2	Embedded HTTP server					
dnsmasq	162.3	Embedded DNS server					
dropbear	307.3	Embedded SSH server	TPLINK	Router	TL-WR940N	V4_160617	MIPSEB
httpd	1692	Embedded HTTP server					
jjhttpd	103.3	Embedded HTTP server	Trendnet	Router	TEW-813DRU	v1(1.00B23)	MIPSEB
lighttpd	327.3	Embedded HTTP server	Netgear	Router	WNAP320	3.0.5.0	MIPSEB

Table 3: IoT programs used for evaluation

identical.

In summary, this evaluation showed that FIRM-AFL can provide transparent emulation as if the program is executing in full-system emulation.

5.2 Efficiency

Benchmark	Augmented mode	User mode	Slowdown
Numeric sort	679.12	686.56	1.08%
String sort	78.36	79.54	1.48%
Bitfield	3.47E+08	3.45E+08	0.00%
FP emulation	163.85	161.72	0.00%
Fourier	1383.6	1,384.00	0.00%
Assignment	20.45	20.75	1.40%
IDEA	4,864.10	4,854.10	0.00%
Huffman	1,749.00	1,743.10	0.00%
Neural Net	1.93	1.95	0.60%
LU Decomp	61.26	61.92	1.00%

Table 4: `nbench` results, the unit is iterations/second. The last column shows the slowdown of augmented mode.

Syscall	Augmented mode	User mode	Overhead
null	0.48	0.48	0.00%
read	0.62	0.60	3.33%
write	0.57	0.52	9.62%
stat	1.31	1.24	5.64%
fstat	0.63	0.61	3.28%
open	2.61	2.50	4.40%
select file	3.52	3.48	1.15%
select tcp	32.74	12.64	159%
pipe(latency)	6.73	6.57	2.44%

Table 5: `lmbench` syscall testing results, the unit is microsecond. The last column shows the overhead of augmented mode.

Standard benchmarks. We evaluated the efficiency of our approach from two angles. First, we evaluated the performance overhead of augmented process emulation using standard performance benchmarks. The result of `nbench` is shown in Table 4. `nbench` is a CPU-bound benchmark suite. On this benchmark, the augmented mode did not impose much overhead, largely due to the fact that these benchmarks are relatively simple, so they do not require many memory synchronization operations and syscall redirection. To evaluate the overhead of syscall redirection, we used the `lmbench`. The result is shown in Table 5. As we can see, for syscalls that are executed locally (e.g., file related syscalls), the overhead is almost negligible. For syscalls that still require redirection (e.g., TCP related), the overhead is much higher.

Fuzzing throughput. In the second performance evaluation, we measured the throughput of FIRM-AFL, under different optimization levels:

(a) *Baseline*: we used TriforceAFL [29] as the baseline. TriforceAFL uses full-system emulation to support fuzzing IoT programs. To avoid rebooting the virtual machine, in this configuration, we added support for QEMU's stock snapshot mechanism (`qemu_savevm` and `qemu_loadvm`) to TriforceAFL. We also use VMI provided by DECAF [23] to capture the precise moment when program is started and terminated.

(b) *Lightweight snapshot*: in this configuration, we changed the snapshot mechanism to our lightweight snapshot (§4).

(c) *Augmented process emulation*: in this configuration, we switched the emulation mode from full-system mode to our augmented process emulation mode (§3).

(d) *Full*: in this configuration, we applied all optimization techniques, including selective syscall redirection.

Figure 3 shows the throughput improvement. Overall, lightweight snapshot boosted the throughput for about 9.3 times (b vs. a). Augmented process emulation boosted the throughput for about 3 times on average (c vs. b). With selective syscall redirection, the throughput had another boost for about 2.9 times on average (d vs. c). So compared with the best result on full-system emulation based fuzzing (b), FIRM-AFL (d) provided an average improvement of 8.2 times.

5.3 Effectiveness of Optimization

In §2.4, we identified three major bottlenecks of full-system emulation: memory address translation, dynamic code trans-

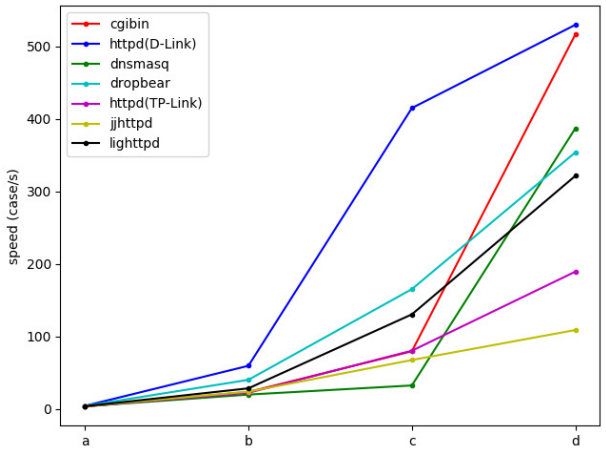

Figure 3: Fuzzing throughput of FIRM-AFL under different optimization level. The x-axis is the optimization level: (a) baseline, (b) w/ lightweight snapshot, (c) w/ augmented process emulation, and (d) w/ selective syscall redirection. Fuzzing throughput for each program is shown in a different color.

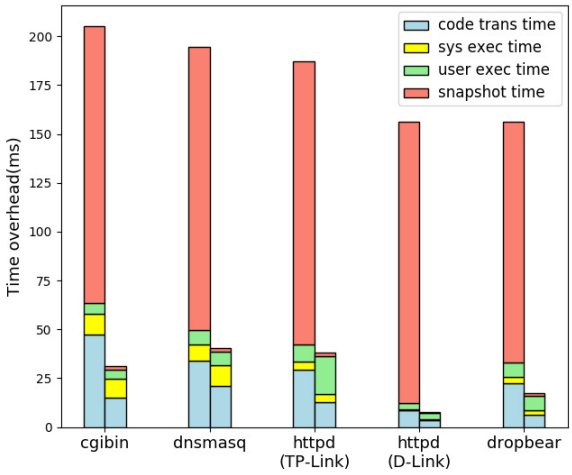

Figure 4: Execution time breakdown: system-mode emulation w/o and w/ lightweight snapshot.

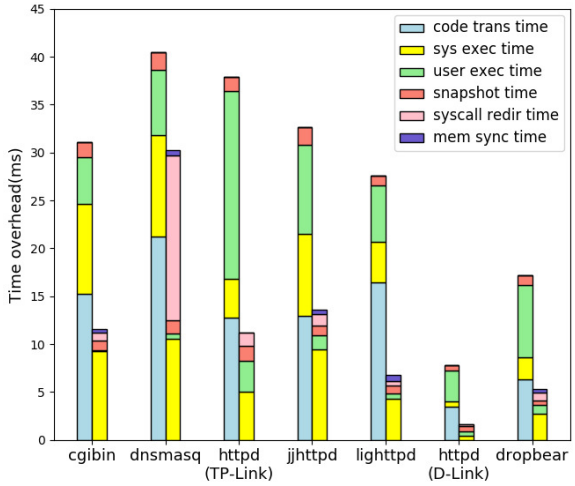

Figure 5: Execution time breakdown: augmented process emulation vs. full-system emulation.

lation, and syscall. In this section, we evaluated whether our optimization techniques successfully addressed these bottlenecks. For this purpose, we break down the total execution time into five parts:

- *User execution time*: the total time spent in executing the logic of the target program, this includes the time spent on software address translation.
- *Memory synchronization time*: in augmented emulation mode, time spent on setup the memory mapping between the user-mode emulator and the full-system emulator.
- *Code translation time*: total time spent on translating the target program.
- *Syscall execution time*: total time spent on system calls in an iteration of execution.
- *Syscall redirection time*: in augmented emulation mode, time spent on redirecting the system call to the full-system emulator.
- *Snapshot time*: the total time spent on storing and restoring memory and CPU states in an iteration of fuzzing. Note that different snapshot mechanisms have different time overhead values. We record the starting and ending time for each page store and restore operations.

Lightweight snapshot. Snapshot overhead only exists for the system-mode emulator. In augmented process emulation, a synchronization mechanism is required to ensure the consistency of snapshot between system and user mode. For these experiments, we measure the snapshot synchronization cost and add it to the snapshot overhead. When comparing the snapshot overhead in Figure 4 and Figure 5, we can see that the lightweight snapshot mechanism leads to more than 100x reduction in the snapshot overhead.

Augmented process emulation. Figure 5 shows the execution time breakdown of full-system emulation and augmented process emulation for the seven IoT programs. The total execution time on average reduces more than 50% except for `dnsmasq`. When analyzing breakdown of execution time, we can see huge reduction on user execution time and code translation time. On average, the user execution time (green bar) was reduced by about 9 times. This is mostly due to the elimination of software address translation. Even if we combine

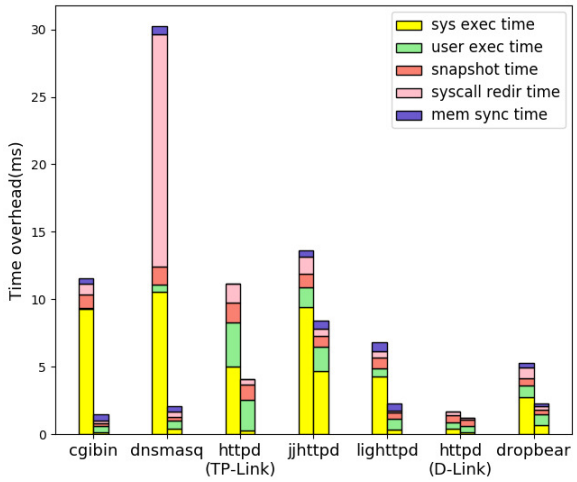

Figure 6: Execution time breakdown: augmented process emulation w/o and w/ selective syscall redirection.

the memory synchronization time (purple bar), the execution time was still reduced by about 5 times.

Another huge reduction is the code translation time. As briefly mentioned in §2, this is due to two optimization techniques. First, when running in full-system mode, QEMU only performs block chaining for basic blocks within the same physical page. This means the emulator has to be invoked to resolve control transfer between pages. In augmented process emulation, QEMU can link any basic blocks as long as they are translated. Second, when using full-system mode for fuzzing, the fuzzer (Triforce) will reset the virtual machine after processing each input. Although we have optimized this step with lightweight snapshot, the code cache will be reset during the restore. This means the same basic block could be translated repeatedly for every fuzzing iteration. In augmented process emulation, we can utilize the code cache pooling technique from AFL to avoid this re-translation. As a result, the amortized code translation time became very small.

Unfortunately, the reduction on user execution time and code translation time is at the cost of increase in overall syscall time, i.e., the combination of syscall execution time and syscall redirection time. In general, the more syscalls the target program issues, the higher the redirection overhead. This is why `dnsmasq` spent significantly more time on syscall redirection than the other programs: it issued more than one thousand system calls which caused more than two thousand state transitions between system mode and user mode. This highlights the necessity of selective syscall redirection.

Selective syscall redirection. Figure 6 shows the execution time breakdown with and without selective syscall redirection. Recall that the goal of redirecting system calls to the full-system emulator is to ensure correct emulation. However, not all system calls require special kernel or hardware support. Therefore, by locally executing system calls that can be fully supported by the host system (e.g., file system related syscalls), we reduce most of the syscall time without jeopardizing correctness. As shown in the figure, after applying this optimization, we observed a huge reduction in system call execution time, because many system calls are now executed by the host OS without address/code translation and device emulation. At the same time, we also observed reduction in syscall redirection time, which has a great impact on programs that issue many syscalls, like `dnsmasq`. A majority of syscalls issued by `dnsmasq` were file operations which can be handled locally by mounting the IoT firmware file system in the host OS. By doing so, the total execution time of `dnsmasq` can be reduced by another 14 times.

To summarize, this evaluation showed that our solutions (augmented process emulation and selective syscall redirection) have successfully addressed the three bottlenecks we identified in §2.4.

5.4 Vulnerability Discovery

In this section, we aim to evaluate how effective FIRM-AFL is in finding vulnerabilities in real-world IoT firmware images.

Data collection. We started with the Firmadyne dataset [4]. We collected these firmware images and tested the emulation condition and network reachability, and then checked the liveness of HTTP and uPnP services by probing their ports. Eventually, we obtained 288 firmware images with active HTTP and uPnP services. We then used getsploit [1] to collect exploits targeting HTTP and UPnP services from online resources, such as exploit-db [3], metasploit [8], and Packet Storm [6]. Then we fed these exploits into the 288 images, and eventually identified 15 exploits that can be launched successfully against 51 firmware images. Table 6 lists these 15 exploits.

We further ran the programs related to these 15 exploits in user-mode QEMU, and observed that only one program `tcapi` that is related to the last five exploits can continue to work in user-mode QEMU. This result once again confirms the necessity of augmented process emulation.

Experiment setup. As our focus in this case study is on fuzzing HTTP and uPnP services, which have well-structured protocol formats. To expedite fuzzing, we made use of the dictionary option "-x" in AFL. We collected keywords for HTTP (from honggfuzz [5]), uPnP and HTTP CGI services (extracted directly from binary programs) respectively. For each service, we then provided a normal service request as the initial seed .

Moreover, to avoid underestimating the performance of full-system emulation with its default snapshot implementation, we enabled lightweight snapshot in it.

Exploit ID	Vendor	Model	Version	Device	Program	Full-System Time to crash	FIRM-AFL Time to crash
CVE-2018-19242	Trendnet	TEW-632BRP	1.010B32	Router	httpd	21h43min	6h2min
CVE-2013-0230	Trendnet	TEW-632BRP	1.010B32	Router	miniupnpd	>24h	9h16min
CVE-2018-19241	Trendnet	TV-IP110WN	V.1.2.2	Camera	video.cgi	19h13min	4h55min
CVE-2018-19240	Trendnet	TV-IP110WN	V.1.2.2	Camera	network.cgi	12h0min	2h21min
CVE-2017-3193	DLink	DIR-850L	1.03	Router	hnap	21h3min	2h54min
CVE-2017-13772	TPLink	WR940N	V4	Router	httpd	>24h	>24h
EDB-ID-24926	DLink	DIR-815	1.01	Router	hedwig.cgi	16h38min	1h22min
EDB-ID-38720	DLink	DIR-817LW	1.00B05	Router	hnap	4h26min	1h29min
EDB-ID-38718	DLink	DIR-825	2.02	Router	httpd	>24h	22h3min
CVE-2016-1558	DLink	DAP-2695	1.11.RC044	Router	httpd	16h24min	2h32min
CVE-2018-10749	DLink	DSL-3782	1.01	Router	tcapi	247s	20s
CVE-2018-10748	DLink	DSL-3782	1.01	Router	tcapi	252s	22s
CVE-2018-10747	DLink	DSL-3782	1.01	Router	tcapi	249s	20s
CVE-2018-10745	DLink	DSL-3782	1.01	Router	tcapi	236s	25s
CVE-2018-8941	DLink	DSL-3782	1.01	Router	tcapi	281s	24s

Table 6: 1-day exploits

The experiments were conducted on a server with 40-core Intel Xeon(R) E5-2687W(v3) 3.10GHz CPU and 125GB of RAM.

Finally, to ensure our evaluation results on fuzzing performance are statistically significant, as suggested by Klees et al. [25], we ran each fuzzing experiment ten instances in parallel for 24 hours. In addition to FIRM-AFL, we also evaluated full system emulation with lightweight snapshot support. We report cumulative number of unique crashes found over time, using `plot_data` in AFL output files.

Evaluation results. We calculate the median time to first crash in full-system emulation and augmented process emulation respectively and record them in the last two columns of Table 6. We can see that FIRM-AFL can find a crash at least 3.6 times faster than full-system emulation, and in many cases more than 10 times faster.

We also plot cumulative number of unique crashes found over time by FIRM-AFL (blue), and fuzzing with full emulation (red) in Figure 7. In each plot, the solid line represents the median result from 10 rounds while the dashed lines represent the lower and upper bounds of 95% confidence intervals for a median. Since last five cases in Table 6 are related to the same program and the results are similar, we just plot the case for CVE-2018-10749 as the representative.

From the result, we can see that in spite of large variations across fuzzing runs, FIRM-AFL was able to find significantly more unique crashes and find them multiple times faster than full emulation. We further investigated these crashes and confirmed that most of these crashes were caused by the same known vulnerabilities. We indeed found two new vulnerabilities, which we will describe next.

0-day vulnerabilities. We discovered two 0-day vulnerabilities using FIRM-AFL, after 7.5 hours and 6 hours respectively. We also tried fuzzing these two programs with full-system emulation using the same initial seeds, and no crash was found within 24 hours. we reported them to IoT manufacturers and MITRE corporation. The details about these two vulnerabilities are described as below.

- CVE-2019-11417: Buffer overflow in Trendnet TV-IP110WN (firmware version: v.1.2.2 build 68). Attackers can exploit the device by using 'languse' parameter in `system.cgi`.
- CVE-2019-11418: Buffer overflow in Trendnet TEW-632BRP (firmware version: v.1.010B32). Attackers can exploit the device by crafting the `soapaction` HNAP interface.

6 Discussion

In this section, we discuss the limitations in our system and shed some light for future work.

Limitation on supported CPU architectures. The current implementation of FIRM-AFL supports the following CPU architectures: mipsel, mipseb and armel, which already account for 90.2% images in the Firmadyne dataset. We expect that supporting more CPU architectures is relatively easy, because the majority of the emulation logic in QEMU is implemented in an architecture-independent manner.

Limitation on supported IoT firmware. Even after more CPU architectures are supported, FIRM-AFL can only fuzz a

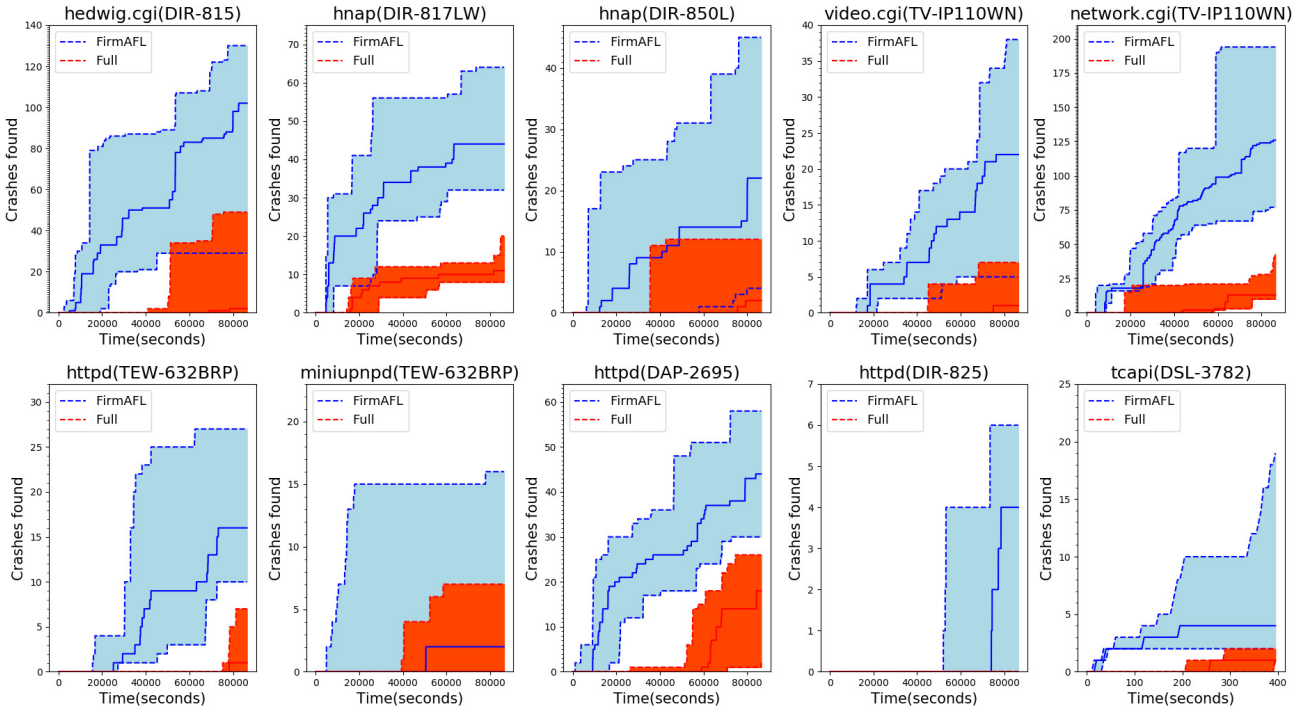

Figure 7: Crashes found over time

program in a firmware image that can be properly emulated by Firmadyne and runs a POSIX-compatible OS (e.g., Linux). This limitation stems deeply from the design of FIRM-AFL, and thus there is no simple solution. An improvement on IoT firmware emulation is orthogonal to this paper. We will leave it for future work. Supporting a non-POSIX program would require a virtualization layer, such that they can run properly within a POSIX process. We are not aware of an existing solution for this. Thus, it can be an interesting future work.

7 Related Work

With the increasing number of IoT devices and their security issues, several techniques are proposed to find the IoT devices vulnerabilities in an automatic manner. These techniques can be categorized into static or dynamic analysis. Lacking the source code of the IoT firmware, static analysis often relies on the binary image and reverse engineering techniques.

Static analysis. Costin et al. presented a large scale analysis of IoT firmware by coarse-grained comparison of files and modules [17]. Their approach is able to find a lot of known bugs within the common third-party projects used by different vendors. Cojocar et al. proposed another approach to heuristically identify parsers and complex processing logics from IoT firmware, and they find several vulnerabilities [16]. That said, these approaches suffer from high false positives and cannot find completely new vulnerabilities. Feng et al. presented a cross-platform bug search technique for firmware images [21]. The technique is based on high-level numeric features comparison, and only takes 0.1 second on average to finish all 154 vulnerabilities searching. Xu et al. further proposed a novel neural network-based approach to detect cross-platform binary code similarity [31]. It can significantly reduce training time and feature vector generation time, as well as improve search accuracy.

Firmalice is another IoT binary analysis framework that employs static analysis techniques [30]. Firmalice utilizes symbolic execution on the firmware binary and uses backward slicing to make the vulnerability analysis tractable. Firmalice focuses only on one slice of the program based on an analyst's specification. The specification provides a clue about the privileged program code. Isolating the potential vulnerable code, Firmalice makes the analysis scalable while also capable of finding new vulnerabilities. That said, Firmalice can only find the authentication vulnerabilities and relies on manual analysis for the slice specification.

Dynamic analysis. On the other hand, dynamic analysis techniques for IoT firmware require either the real devices or an emulation of some sort. Black-box fuzzing is a common approach to discover vulnerabilities by directly interacting with devices. Recently, several works have developed dynamic emulators for IoT devices. For example, Zaddach et al. developed a dynamic analysis framework for IoT devices by redirecting hardware requests from the emulator to

the actual hardware [33]. Based on it, Marius et al. developed a dynamic multi-target orchestration framework that can enable interoperability between different dynamic binary analysis framework, debuggers, emulators and real physical devices [27]. However, the large number of hardware limits its scalability, and also imposes a large overhead.

Chen et al. proposed a robust software-based full system emulation. Their emulation is based on kernel instrumentation [13]. Their goal is to perform automatic vulnerability verification that has no ability to find unknown vulnerabilities. Both Avatar and Firmadyne do not use techniques such as fuzzing that are capable of finding completely new vulnerabilities in real applications. Anderi et al. conducted dynamic analysis to achieve automated vulnerability discovery within embedded firmware images [18]. The tool aims at discovering web-interface related vulnerabilities by using web pentesting tools. However, it cannot find vulnerabilities of other modules in IoT firmware.

IoT fuzzing. For IoT fuzzing, and closest to our work, Muench et al. developed six live analysis heuristics including call stack tracing and call frame tacking [28]. Muench et al. built their system on top of Avatar [33] and PANDA [19], and their system can effectively detect memory corruption for IoT devices. However, this system takes target systems as black-box and feeds input from outside which imposes overhead on the devices startup and rebooting for each fuzzing session. Further, unlike greybox fuzzing, the input space exploration is very blind, and hence the chance of finding a bug is very low. In our work, we utilize greybox fuzzing, and aim to minimize each fuzzing iteration overhead so that the fuzzer can test more test cases in the same unit of time. In addition, Alimi et al. proposed to use fuzzing techniques and specific simulators (JCOP) to discover vulnerabilities in programs hosted into smart cards [11]. The methodology does not scale due to emulation problems of various kinds of IoT firmware.

8 Conclusion

Coverage-based greybox fuzzing has proven to be an effective way to find vulnerabilities in real-world programs. Yet, applying greybox fuzzing to IoT firmware has not been realized due to two main challenges. Firstly, state-of-the-art greybox fuzzers like AFL fail to run many IoT programs due to specific hardware dependencies. Secondly, solutions that can tackle the first challenge (e.g., by employing full-system emulation) yield very low throughput. We proposed a novel technique, augmented process emulation to address both challenges at the same time. With augmented process emulation, we achieve high throughput fuzzing by running the target program in a user-mode emulator and switch to a full-system emulator when the target program invokes a system call that has specific hardware dependencies.

We evaluated the transparency and the efficiency of FIRM-AFL, our prototype implementation of greybox IoT fuzzing based on the augmented process emulation. The results showed that our system is transparent and its throughput outperforms all the state-of-the-art IoT firmware fuzzers by one order of magnitude. Our case study further showed that FIRM-AFL could indeed find both 1-day vulnerabilities much faster than full-system emulation and was able to find two new vulnerabilities within only two hours on a single machine.

Acknowledgement

We thank our shepherd Dr. Yongdae Kim and the anonymous reviewers for their insightful comments on our work. This work is partly supported by Key Program of National Natural Science Foundation of China under Grant No. U1766215, National key R&D Program of China under Grant No. 2016YFB0800202, Strategic Priority Research Program of Chinese Academy of Sciences under Grant No. XDC02020500, International Cooperation Program of Institute of Information Engineering, CAS under Grant No. Y7Z0451104, National Science Foundation under Grant No. 1664315, Office of Naval Research under Award No. N00014-17-1-2893, Guangdong Province Key Area R&D Program of China under Grant No. 2019B010137004. We also thank the support provided by China Scholarship Council (CSC) for Yaowen Zheng's visiting to UCR. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Command line utility for searching and downloading exploits. https://github.com/vulnersCom/getsploit.

[2] The cyber grand challenge. http://blogs.grammatech.com/the-cyber-grand-challenge.

[3] Exploit database - exploits for penetration testers, researchers, and ethical hackers. https://www.exploit-db.com/.

[4] Firmadyne datasheet. https://cmu.app.boxcn.net/s/hnpvf1n72uccnhyfe307rc2nb9rfxmjp/folder/6601681737.

[5] honggfuzz. a security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options. http://honggfuzz.com/.

[6] Information security services, news, files, tools, exploits, advisories and whitepapers. https://packetstormsecurity.com.

[7] LMbench - tools for performance analysis. http://www.bitmover.com/lmbench/.

[8] Metasploit | penetration testing software, pen testing security. https://www.metasploit.com.

[9] nbench. https://www.math.utah.edu/~mayer/linux/bmark.html.

[10] Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016, gartner. http://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016, February 2017.

[11] V. Alimi, S. Vernois, and C. Rosenberger. Analysis of embedded applications by evolutionary fuzzing. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 551–557, July 2014.

[12] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[13] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Network and Distributed System Security Symposium, NDSS*, February 2016.

[14] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Networked and Distributed System Security Symposium (NDSS'18)*, February 2018.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[16] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. PIE: Parser identification in embedded systems. In *Annual Computer Security Applications Conference (ACSAC'15)*, December 2015.

[17] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, August 2014.

[18] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, May 2016.

[19] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, 2015.

[20] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy*, May 2016.

[21] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, 2016.

[22] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[23] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *International Symposium on Software Testing and Analysis (ISSTA'14)*, July 2014.

[24] Pereyda J. boofuzz. https://github.com/jtpereyda/boofuzz, 2016.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, October 2018.

[26] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.

[27] Marius Muench, Aurélien Francillon, and Davide Balzarotti. Avatar^2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (BAR'18)*, February 2018.

[28] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Network and Distributed System Security Symposium (NDSS'18)*, February 2018.

[29] NCC-Group. TriforceAFL. https://github.com/nccgroup/TriforceAFL, 2017.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Network and Distributed System Security Symposium (NDSS'15)*, February 2015.

[31] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, October 2017.

[32] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security Symposium*, August 2018.

[33] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS'14)*, February 2014.

[34] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.