Efficient Greybox Fuzzing of Applications in Linux-Based IoT Devices via Enhanced User-Mode Emulation

Yaowen Zheng[†]
Continental-NTU Corporate Lab,
Nanyang Technological University
Singapore
yaowen.zheng@ntu.edu.sg

Yuekang Li[‡]
Continental-NTU Corporate Lab,
Nanyang Technological University
Singapore
yuekang.li@ntu.edu.sg

Cen Zhang
Continental-NTU Corporate Lab,
Nanyang Technological University
Singapore
CEN001@e.ntu.edu.sg

Hongsong Zhu
Beijing Key Laboratory of IoT
Information Security Technology, IIE,
CAS; School of Cyber Security, UCAS
Beijing, China
zhuhongsong@iie.ac.cn

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

Limin Sun[‡]
Beijing Key Laboratory of IoT
Information Security Technology, IIE,
CAS; School of Cyber Security, UCAS
Beijing, China
sunlimin@iie.ac.cn

ABSTRACT

Greybox fuzzing has become one of the most effective vulnerability discovery techniques. However, greybox fuzzing techniques cannot be directly applied to applications in IoT devices. The main reason is that executing these applications highly relies on specific system environments and hardware. To execute the applications in Linux-based IoT devices, most existing fuzzing techniques use full-system emulation for the purpose of maximizing compatibility. However, compared with user-mode emulation, full-system emulation suffers from great overhead. Therefore, some previous works, such as Firm-AFL, propose to combine full-system emulation and user-mode emulation to speed up the fuzzing process. Despite the attempts of trying to shift the application towards user-mode emulation, no existing technique supports to execute these applications fully in the user-mode emulation.

To address this issue, we propose EQUAFL, which can automatically set up the execution environment to execute embedded applications under user-mode emulation. EQUAFL first executes the application under full-system emulation and observe for the key points where the program may get stuck or even crash during user-mode emulation. With the observed information, EQUAFL can migrate the needed environment for user-mode emulation. Then, EQUAFL uses an enhanced user-mode emulation to replay system calls of network, and resource management behaviors to fulfill the needs of the embedded application during its execution.

We evaluate EQUAFL on 70 network applications from different series of IoT devices. The result shows EQUAFL outperforms the state-of-the-arts in fuzzing efficiency (on average, 26 times faster than AFL-QEMU with full-system emulation, 14 times than Firm-AFL). We have also discovered ten vulnerabilities including six CVEs from the tested firmware images.

[†]Also with Beijing Key Laboratory of IoT Information Security Technology, IIE, CAS.
[‡]Corresponding Author(s)

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; **• Security and privacy → Embedded systems security**; **Software and application security**.

KEYWORDS

Greybox Fuzzing, Linux-based IoT Devices, Enhanced User-mode Emulation

ACM Reference Format:
Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 2022. Efficient Greybox Fuzzing of Applications in Linux-Based IoT Devices via Enhanced User-Mode Emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534414

1 INTRODUCTION

With the rapid development of the Internet of Things (IoT), billions of IoT devices are connected to the Internet. Compared with traditional IT vendors, vendors of IoT devices pay more attention to system functionalities, rather than system security. As a result, various unknown vulnerabilities exist for the application running in the IoT systems, providing the attackers with a large number of attack surfaces. For example, in late 2016, the Mirai virus was developed to exploit the vulnerability of Linux-based IoT devices that launches a massive DDoS attack on the east coast of the US [35]. This event has brought a significant impact on cyberspace security. In summary, security researchers need to find vulnerabilities inside the Linux-based IoT devices as quickly as possible, especially for those applications which serve as gates between IoT devices and the Internet.

Greybox fuzzing is a practical testing technique used for finding vulnerabilities in software. The basic idea of greybox fuzzing is to use light-weight program instrumentation to collect execution feedback of the program under test (PUT), such as code coverage, to guide the entire testing process. Although greybox fuzzing has shown good performance on conventional programs running in desktop environment, it cannot be directly applied to applications running in embedded devices. This is mainly because of the lack of system and hardware support for executing the applications. To solve this problem, existing greybox fuzzing techniques use emulation techniques to execute the embedded applications [34, 51].

Existing emulation techniques, such as QEMU, supports both user-mode emulation and full-system emulation. Compared with full-system emulation, user-mode emulation enjoys much smaller execution overhead at the cost of compatibility due to the inability of emulating the system calls and execution context. Therefore, most existing greybox fuzzing techniques use full-system emulation to execute the embedded applications. However, this approach suffers from efficiency problems due to the heavy overhead of emulating an entire system. To improve the efficiency, a recent work, Firm-AFL, proposes a technique to smartly switch between user-mode emulation and full-system emulation. The mechanism of Firm-AFL is that it allows the application to run in user-mode as long as it is not executing a system call and whenever the application meets a system call, Firm-AFL will switch to full-system emulation to acquire the system call result. Compared with full-system emulation, Firm-AFL has improved execution speed, but the improvement becomes negligible when the PUT involves a lot of system calls. Therefore, the challenge now is, can we execute the embedded applications fully under user-mode emulation without sacrificing too much compatibility?

To address the challenge for fuzzing applications in Linux-based IoT devices, we present EQUAFL, a greybox fuzzing framework augmented by enhanced user-mode emulation, which enjoys both high efficiency and high compatibility. The basic mechanism of the enhanced user-mode emulation is to automatically set up the execution environment so that the system calls of the PUT can be directly passed to host machine. In this way, the PUT can fully execute in user-mode emulation, avoiding the overhead incurred by the emulation of system calls. In order to automatically set up the execution environment for the PUT, EQUAFL uses an *observe-replay* strategy. First, EQUAFL executes the PUT with full-system emulation and observes for key behaviors of the system such as setting the launch variables, generating the configuration files, network manipulation and so on. Then, EQUAFL will replay the observed system behaviors to set up the execution environment of the PUT. Different system behaviors require different observation and replay methods and the two most complicated behaviors are dynamic configuration file generation and network interaction. The former one requires process awareness during observation and filesystem synchronization during replay. The latter one requires state aware observation and replay.

We evaluate both the compatibility and efficiency of EQUAFL on 70 real-world applications in Linux-based IoT devices. The result shows that EQUAFL outperforms both AFL-QEMU with full-system emulation (26 times faster in average) and Firm-AFL (14 times faster in average) in terms of execution speed. During the experiments, EQUAFL discovers ten previously unknown vulnerabilities (including six CVEs) in eighteen embedded devices, proving its usefulness in real-world vulnerability discovery.

Contributions. In summary, we make the following contributions:

- We proposed a novel technique called EQUAFL which can automatically set up the execution environment to emulate embedded programs fully in user-mode. The enhanced user-mode emulation of EQUAFL can guarantee both high compatibility and high efficiency.
- We implemented EQUAFL as a coverage-guided greybox fuzzing framework based on AFL and QEMU.
- We extensively evaluated EQUAFL on 70 real-world network applications in different product series from major embedded device vendors. The result demonstrated the high compatibility and efficiency of EQUAFL. Also, we found ten 0-day vulnerabilities including six CVEs with EQUAFL.
- We released the source code of EQUAFL at https://github.com/zyw-200/EQUAFL to facilitate future research.

2 BACKGROUND

2.1 Emulation-Based Fuzzing

QEMU [2] is a generic machine emulator for various CPU architectures. It supports two modes of emulation: full-system emulation and user-mode emulation. For IoT systems, full-system emulation emulates the whole system of the embedded firmware, which includes the kernel, drivers, and applications. On the contrary, user-mode emulation only emulate an individual Linux-based applications in firmware by delegating the system calls to the host machine. Therefore, comparing with full-system emulation, the advantage of user-mode emulation is higher execution efficiency because the overhead of emulating the whole system can be very high, while the disadvantage of user-mode emulation is the lack of compatibility because if the applications requires some resources or system calls not supported by the host machine, then the user-mode emulation will fail.

Fuzzing is one of the most effective software vulnerability discovery techniques. AFL is a widely used coverage-guided greybox fuzzer which supports to use QEMU as the emulation engine for the PUT execution.

AFL + QEMU user-mode emulation. By default, AFL uses user-mode QEMU as its emulator. Since in user-mode emulation, the execution of system calls are forwarded directly to the host machine, the execution of the PUT may lead to unexpected states such as crash, hang, etc, if the host machine is not compatible with the PUT. For example, when we run a application (sbin/httpd) from D-Link firmware (TRENDNet TEW-634GRU series) in AFL, it would report the message "/var/run/httpd.pid: No such file or directory" and then exit quickly without running into the deep program states. Although the fuzzing process is still going on, it only tests the limited code paths of the application. To explore the failure reasons, we utilize user-mode QEMU to test 70 network applications in our evaluation dataset (§6), and finally summarize the failure reasons as follows.

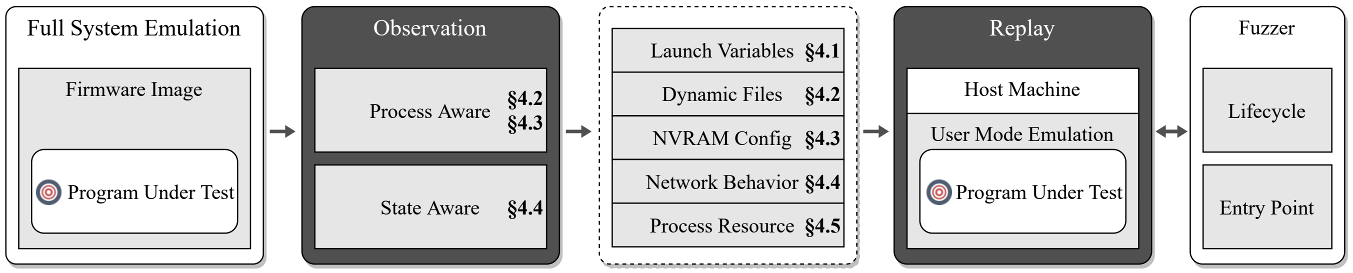


Figure 1: The workflow of EQUAFL

- **Wrong launch variables.** Some PUTs terminate at the beginning of the execution because of incorrect or even no launch variables are provided.
- **Missing dynamically generated files.** Some PUTs may require some files such as configuration files before execution. In IoT devices, most of the such files are generated dynamically during the booting of the device. Missing these files will cause the PUT to terminate early.
- **Inconsistent NVRAM configurations.** NVRAM is a type of flash memory, which is widely used in IoT devices to store the configurations. If the configurations stored in the NVRAM of the IoT device and the host machine conflict with each other, the PUT will execute with errors.
- **Inconsistent network behaviors.** Most of the PUTs for IoT devices need to interact with users through the network. In the cases where the host machine cannot provide proper network interactions, the PUT will hang or terminate early.
- **Inconsistent process resource limits.** Linux system uses process resource limit to prevent the over-consumption of specific system resources by processes. In some cases, the value of process resource limit on the host machine is much higher than that of the emulated firmware, which may prevent the PUT from executing efficiently on the host machine.
- **Lack of hardware.** In some cases, the PUT requires specific hardware to execute and absence of the required hardware will cause execution failures.

In summary, fuzzing IoT programs with AFL + QEMU user-mode emulation suffers severe compatibility issues and all the causes of the execution failures, except for the lack of hardware, can be avoided by using full-system emulation.

AFL + QEMU full-system emulation. AFL integrated with full-system emulation of QEMU can support the fuzzing of entire firmware images and applications in it. However, since QEMU adds a virtualization layer to support the whole firmware emulation, the fuzzing speed with full system emulation can be pretty slow. Therefore, fuzzing IoT applications with AFL + QEMU full-system emulation is not viable in practice.

AFL + QEMU hybrid emulation. To solve the challenges from above two approaches, Firm-AFL [51] attempted to combine both full-system emulation and user-mode emulation to support fuzzing of Linux-based IoT applications. It executes the user-space code in user-mode emulation and redirect the system calls to full-system emulation, which guarantees both compatibility and efficiency of the fuzzing process to a certain degree.

However, Firm-AFL still suffers from efficiency problem when the application has frequent system call operations such as file read and write. If a large number of files required by applications are generated during the firmware initialization, and cannot be found in the original filesystem, Firm-AFL would frequently redirect the system call execution from user-mode emulation to full-system emulation, which greatly slow down the execution speed. Therefore, the efficiency of Firm-AFL still has room for improvement.

2.2 Terminology

Guest/Host Machine. In full-system emulation, the guest machine refers to the virtual machine containing the initialized firmware system running in the emulator. The host machine refers to the operating system that is hosted on the physical machine (here we ignore the case where the host machine is also a virtual machine). The emulator of the guest machine runs in the host machine.

Page Global Directory. Page Global Directory (PGD) refers to the top physical page frame of a process, which is widely adopted in the modern operating system. The starting address of PGD is unique among numerous processes, so that we can use it to identify a user-space process in Linux-based system. We utilize the PGD in §4.2 to achieve file-related observation with process awareness.

3 OVERVIEW

The goal of our system is to enable efficient greybox fuzzing of Linux-based network applications in IoT devices. As discussed in §2.1, current fuzzing works suffer from either performance or compatibility problems. On the one hand, AFL with user-mode emulation ensures the high fuzzing speed, but results in low compatibility. On the other hand, AFL with full-system emulation emulates more target applications successfully but lacks efficiency. To this end, we propose to combine full-system emulation and user-mode emulation in an innovative manner to develop a fuzzer that satisfies two requirements:

- High compatibility: Applications should behave the same as in full-system emulation.
- High efficiency: The speed of fuzzing should be as fast as possible.

The fuzzer we proposed is called EQUAFL, which is an AFL-based framework through Enhanced QEMU User-mode emulation. Figure 1 shows the workflow of EQUAFL, which consists of two major steps: observation and replay. In the observation stage, EQUAFL executes the PUT with full system emulation and observes the key behaviors of the system. According to the failure reasons discussed in §2.1, the key behaviors of the system include setting of the launch variables, file generation, NVRAM related operation, network interaction, and process resource limits. Among these behaviors, the two most complicated behaviors are dynamic configuration file generation and network interaction. Therefore, we propose to observe the dynamic configuration file generation and NVRAM configurations with process awareness (§4.2 and §4.3). Meanwhile, we propose to observe the network behaviors with state awareness (§4.4). Besides, we use several heuristics to observe other information such as launch variables and process resource limits (§4.1 and §4.5). In the replay stage, EQUAFL carries out the replay by either deploying system resources such as dynamic configuration files on the host machine or performing the interception of system calls execution during the user-mode emulation. After the observation and replay of key behaviors in the emulated system, the PUT can fully execute in the enhanced user-mode emulation, avoiding the overhead incurred by delegating the system call execution to the full-system emulation. Last but not least, we integrate the enhanced user-mode emulation with the AFL fuzzer by adjusting the PUT lifecycle management and entry point of fuzzing so that EQUAFL can fuzz IoT applications which receive inputs from the network.

4 APPROACHES

4.1 Launch Variables Settlement

Launch variables refer to both arguments and environment variables that the PUT starts with in the emulated system. The PUT may terminate early without correct launch variables, thus we need to settle the launch variables when executing the PUT in the user-mode emulation. Here we denote the PUT as p^* [1], and its program name, arguments and environment variables as p^*_{name}, p^*_{vars} and p^*_{envs}. Generally, these variables are stored within embedded firmware in diverse ways, including written in configuration files, hard-encoded in binary executables, and even passed by the parent process. Thus, it is hard to practically extract these parameters via static approaches. Instead, we propose to obtain concrete values of p^*_{vars} and p^*_{envs} by performing both static pattern analysis for Linux kernel and run-time analysis during full-system emulation.

Observation. Based on the fact that Linux kernel invokes `execve` system call when starting a new program, we dump p_{name}, p_{vars} and p_{envs} of the newly starting program by instrumenting the kernel function `do_execve` during the full-system emulation. Since `do_execve` is a common architecture part of the Linux kernel, it is safe to make an assumption that most Linux-based firmware will not modify that code. Therefore, by analyzing the source code of Linux kernel, we summarize a binary pattern that can be used to locate the exact assembly instruction for dumping these launch variables when emulating the firmware. Specifically, we find three function call instructions (one calls `copy_strings_kernel` and another two call `copy_strings`), which contain the registers that can be used to calculate the addresses of p_{name}, p_{vars} and p_{envs}. Then, we find a nearest successor basic block from these instructions where QEMU can instrument. Finally, we dump the values at that basic block during the system emulation. Specifically, the basic block we choose is exactly the entry point of the function `search_binary_handler`.

[1]In the following of this section, we use the asterisk symbol to indicate a variable is used for replay.

Replay. After the observation, we obtain a collection of (p_{name}, p_{vars} and p_{envs}), which can be used to represent different processes during the firmware emulation. We then recognize p_{vars} and p_{envs} as the target p^*_{vars} and p^*_{envs}, where p_{name} equals to p^*_{name}. In the fuzzing process, we utilize user-mode emulation to execute p^* with p^*_{vars} and p^*_{envs}. Eventually, the PUT can run into deep states in user-mode emulation for further testing.

4.2 Filesystem State Synchronization

In full-system emulation, many firmware images mount a temporary filesystem and constantly change the filesystem state during the initialization phase. On the host machine, we cannot update filesystem state without initialization of firmware. As a result, the PUT would execute to unexpected states in user-mode emulation due to improper filesystem state of the host machine. For example, firmware dynamically generate files such as configuration files during the initialization phase [5]. These files are not found in the original filesystem of firmware. Thus, the PUT may execute to error states without accessing the specific configuration file on the host machine.

To this end, we utilize *observe-replay* strategy to synchronize the filesystem state between the guest machine and the host machine. Specifically, we attempt to observe file-related system call execution in the guest machine. For each observed file-related system call execution, we re-execute it on the host machine. We constantly repeat such observe-replay action until we detect that the PUT has started to run. Until then, the filesystem state for the PUT execution is fully set up. Unfortunately, when taking replay action on the host machine, some arguments of file-related system calls are unknown. Fox example, as shown in the right hand of Figure 2, when we re-execute the `write` system call, we cannot specify the value of fd^*_{host} directly. To solve this problem, we propose a process-aware observation approach to build the mapping for files between the guest machine and the host machine.

Accurate Process Identification. To achieve the process-aware observation, we first propose to identify the current executing process in the guest machine. The workflow of the process identification consist of two steps: process collection and process inference. In the process collection step, we constantly update the information of all the running processes during the firmware emulation. Specifically, since both `fork` and `execve` system calls take part in creating a new process, we instrument at the end of these two system call execution to gather the information of newly generated process. At the instrumentation point, we traverse `task_struct` data structure (the process descriptor in Linux kernel), and find newly generated one that represents new process. For each process, we obtain the information from `task_struct` including the starting address of page global directory (PGD), process identifier (PID) and PID of the parent process (PPID). Finally, the set *Collection* that contains information of all the running processes is produced as shown in Equation 1. In the process inference step, we acquire the PGD value of the current executing process p by monitoring the specific register or memory regions. For example, for the ARM architecture, we obtain the PGD value of p by accessing the specified register of

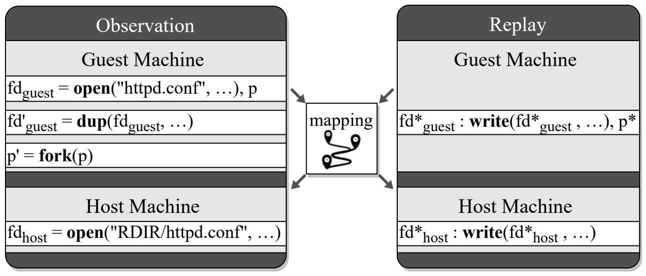

Observation
Guest Machine
fd_guest = open("httpd.conf", …), p
fd'_guest = dup(fd_guest, …)
p' = fork(p)
Host Machine
fd_host = open("RDIR/httpd.conf", …)
mapping
Replay
Guest Machine
fd*_guest : write(fd*_guest , …), p*
Host Machine
fd*_host : write(fd*_host , …)

Figure 2: Example of process-aware observation

system control co-processor (CP15). After obtaining the PGD value, we further search *Collection* to find the matching item where the value of PGD are equal. We obtain the PID and PPID of p, which will be used for the filesystem state synchronization between the guest machine and the host machine.

$$Collection \leftarrow \{(t_{pgd}, t_{pid}, t_{ppid}) | t \in task\_struct\} \tag{1}$$

Process-aware Observation. With the ability to identify the current executing process in the guest machine, we start to observe the file-related system call execution with process awareness. Initially, we filter the file-related system calls that do not affect file states or modify the file contents (e.g., read). Then, we classify the rest of the file-related system calls into two types based on their arguments as shown in Table 1: one type is to directly handle the file path; another type is to handle the the file descriptor. For the latter type, we cannot easily get the corresponding file descriptor on the host machine. Therefore, we propose to observe the relation of files with process awareness as illustrated in Figure 2. First, we monitor the system call that creates a file (e.g., open) and get the returned file descriptor fd_{guest}. Also, we identify current executing process p that executes the system call. Meanwhile, we re-execute the same system call on the host machine, and get the returned file descriptor fd_{host}. Since the firmware may create a copy of file descriptor by executing dup-like system calls, we add the associated guest file descriptors into the set FD_{guest} as shown in Equation 2. Since Linux is multi-task operating system, file descriptors can be inherited by child processes to take part in the modification of files. Thus, we associate the processes and add them to P as shown in Equation 3. After that, we create the mapping M as shown in Equation 4, which builds up the relation between P, FD_{guest} and fd_{host}. That means, in the subsequent execution, if the process belongs to P, and it executes other file-related system calls on a file descriptor that belongs to FD_{guest}, we re-execute the system call on fd_{host} on the host machine.

$$\begin{gathered}\Delta FD_{guest} \leftarrow \{\forall fd' \mid fd' = DUP(fd) \land fd \in FD_{guest}\} \\ FD_{guest} \leftarrow FD_{guest} \cup \Delta FD_{guest}\end{gathered} \tag{2}$$

$$P \leftarrow \{\forall p' \mid p'_{pid} = p_{pid} \lor p'_{ppid} = p_{pid}\} \tag{3}$$

$$M : P \times FD_{guest} \longmapsto fd_{host} \tag{4}$$

Replay. For each file-related system call invoked in the guest machine, we re-execute it on the host machine to synchronize the filesystem state. Generally, we show the overall procedure of file system call execution replay in Algorithm 1. The inputs of algorithm include (1) id_{sys}: the system call we are replaying; (2) p^*: the process that executes the system call id_{sys} in the guest machine; (3) obj_arg: the argument of id_{sys} that refers to the file object. It can be a file path or a file descriptor. (4) $other\_args$: other arguments of id_{sys} except obj_arg; (5) $RDIR$: the absolute directory path of extracted firmware filesystem on the host machine.

Table 1: File-related system calls

Argument Type	System Calls
File paths	mount, mkdir, rmdir, mkdirat, link, symlink, unlink
File descriptor	open, read, write, dup, dup2, dup3, create, fcntl, pipe

Algorithm 1: Replay of File-related System Call Execution

```
def Replaying($id_{sys}, p^*, obj\_arg, other\_args, RDIR$):
    if Type($obj\_arg$) = string then
        $path_{guest} \leftarrow obj\_arg$;
        if $path_{guest}$.startwith('/') = true then
            $path_{host} \leftarrow$ strcat($RDIR, path_{guest}$);
        else
            $path_{host} \leftarrow path_{guest}$;
        if Type($path_{guest}$) = symbolic_link then
            $path_{source} \leftarrow$ get_source($path_{guest}$);
            if $path_{source}$.startwith('/') = true then
                $path_{source} \leftarrow$ strcat($RDIR, path_{source}$);
        do_syscall($id_{sys}, path_{host}, other\_args$);
    else
        $fd^*_{guest} \leftarrow obj\_arg$;
        $fd^*_{host} \leftarrow$ M($p^*, fd^*_{guest}$);
        do_syscall($id_{sys}, fd^*_{host}, other\_args$) ;
```

The replay procedure of file-related system call execution can be divided into two parts. For the system calls that handle a file path $path_{guest}$, we infer a new file path $path_{host}$ to assure that the replay can be restricted in the extracted firmware directory instead of the root directory on the host machine. If $path_{guest}$ is an absolute path, we add $RDIR$ in front of $path_{guest}$ to generate $path_{host}$. If $path_{guest}$ is a symbolic link and its source $path_{source}$ is an absolute path, we add $RDIR$ in front of $path_{source}$. After that, we execute the system call on $path_{host}$ (line 2 – 12). On the other hand, for the file-related system call that handle a file descriptor, we identify the current executing process p^*, and utilize the mapping M as built via Equation 4. Based on it, we find out fd^*_{host}, and finally execute the system call on it in the host machine (line 14 – 16).

4.3 NVRAM Configuration

We utilize *observe-replay* strategy to generate the run-time NVRAM configurations on the host machine.

Observation. In the state-of-the-art full-system emulation techniques (e.g., FIRMADYNE [5]), regular files are allocated to store the data of NVRAM configuration. Thus, the emulation of NVRAM access is implemented by redirecting related APIs to the data access in such regular files. Since we have achieved filesystem state synchronization on the host machine as elaborated in §4.2, the files that stored the data of NVRAM configuration are also generated on the host machine.

Replay. In the user-mode emulation, we perform the replay by redirecting the NVRAM access of the PUT to the NVRAM configuration files on the host machine. Specifically, we utilize `LD_PRELOAD` technique to execute the PUT with a customized library, which overwrites all the NVRAM-related APIs including `set`, get, `commit` and `clear` NVRAM operations.

4.4 Network Behavior

Unlike other system behaviors, network behaviors can be affected by the network interaction with the outside world. For example, due to different occasions that the user send the request to the PUT, the network-related system call sequences would vary widely in different runs. To emulate the network behavior, we first learn the network state machine from plenty of observed network-related system call sequences. Then, we emulate the network behavior of the PUT based on the state machine.

State-aware Observation. First, we automatically collect network-related system call sequences of the PUT during full-system emulation. Since applications in Linux system use socket not only for the network communication but for the inter-process communication, we identify the network-related socket by monitoring *type* parameter of `socket` system call. If *type* is identical to `AF_INET6` or `AF_INET`, we identify the generated `socket` as a network-related descriptor fd_{net}. Later on, the system calls executing on fd_{net} are identified as network-related system calls. After we collect network-related system calls sequence, we carry out the state machine as shown in Figure 3 to guide the emulation of network behaviors. Specifically, the Linux kernel issues a series of common network-related system calls to support network communication. Most of the network-related system calls including `bind`, `listen`, `accept`, `read/recv/recvfrom` are executed in a certain order. Otherwise, network-related applications could not achieve network communications correctly.

Furthermore, I/O multiplexing operations are frequently used in our network applications to monitor the network sockets whether they are ready to read or write. Representative system calls consist of `Poll`, `Select`. Unlike typical socket operation, they are not deemed necessary to implement the network communication. From our observation, we found that the usage of I/O multiplexing shows a variety of forms in different applications. Fortunately, the network state transition process can also be modeled as shown in Figure 3: (1) When the application executes to `Poll` on fd_{net} at the first time, fd_{net} is ready for connection, and ready state is set on fd_{net}. After that, the newly network descriptor fd'_{net} is accepted. (2) When the application executes to `Poll` on fd'_{net}, the ready state is set on fd'_{net}. After that, fd'_{net} is waiting for the data. (3) When the application executes to `Poll` at the third time, fd'_{net} has received the data, the current fuzzing iteration can terminate and then run into the next fuzzing round in most cases.

Replay. In the replay phase, we follow the state machine in Figure 3 to emulate the network-related system calls. Specifically, we instrument at the beginning of network-related system calls, and then feed the expected result as it has executed successfully. We also maintain the network resources such as `sock_addr` data structure

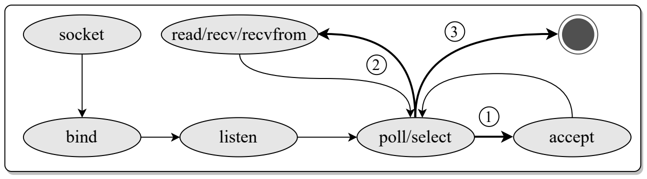


Figure 3: Example of network system call sequences

by filling the corresponding memory, which ensures that subsequent execution relying on these network-related resources can execute correctly.

4.5 Process Resource Limits

Process resource limit refers to the maximum number of specific kernel resource that the application can consume, and improper setting of resource limits may affect the behavior of the PUT. For example, the resource limit `RLIMIT_NOFILE` specifies the maximum number of file descriptors that can be opened by the PUT. In the primary execution stage of some PUTs, they may traverse all system descriptors, the number of which ranges from zero to the `RLIMIT_NOFILE` limit. As a result, the setting of large `RLIMIT_NOFILE` limit value would slow down the execution of the PUT. In fact, due to different system performance, the value of `RLIMIT_NOFILE` in kernel of the host machine is much higher than that in the embedded firmware. Thus, we cannot directly use the value of resource limit from the host machine.

Therefore, we utilize observe-replay strategy to set the process resource limits for the PUT in user-mode emulation. Note that the Linux kernel provides `setrlimit` and `getrlimit` system calls to set or get values of process resource limits. Thus, we retrieve process resource limit values by monitoring `getrlimit` during full-system emulation of firmware in advance. After that, when the PUT requires the resource limit through `getrlimit` at the first time, we provide the observed values directly to them.

5 IMPLEMENTATION

5.1 Emulation

Observation. Initially, EQUAFL executes the PUT with full-system emulation by utilizing FIRMADYNE, which is an automated and scalable full-system emulation platform for Linux-based embedded firmware [5]. During the full-system emulation, EQUAFL performs observation by instrumenting full-system mode of QEMU at the end system calls execution. Specifically, when the firmware executes to a system call. QEMU will handle the specific exception or interrupt invoked by the system. EQUAFL treats it as the starting point of system call execution, and record current execution context. Later, the firmware will execute to the kernel space, and EQUAFL instruments at the end of each basic block to detect whether the firmware execution returns back to the user-space. When firmware executes to the user-space and the execution context are equal to the previous records, EQUAFL treats it as the end of the system execution. Eventually, we can collect the argument and the return value of each system call execution in the full-system emulation.

Table 2: Real-world dataset

Vendor	Product series	Device types	Samples
D-Link	DIR / DI DAP / DSL DSP / GO-RT	Router Range extender Smart plug	30
TRENDnet	TEW-DRU / TV-IP TEW-AP / TEW-DRE	Camera / Router Access point Range extender	11
NETGEAR	WNDR / JNR XWN / WNCE EX / XAVN	Router / Access point Wi-Fi adapter / Extender	29

Replay. EQUAFL conducts replay from two aspects. For the launch variables, filesystem state synchronization, NVRAM configuration, EQUAFL implements the replay mechanism by deploying the related resource directly on the host machine, which can be used by the PUT in the user-mode emulation based fuzzing. For network behavior, process resource limits, EQUAFL implements the replay strategy by instrumenting the user-mode mode of QEMU. In the phase of launch variables settlement, we find the exact point to dump launch variables by utilizing IDA Pro [21]. Besides, in the phase of filesystem state synchronization, we first utilize `Binwalk` utilities [25] to unpack the firmware and get the original filesystem. After the filesystem state synchronization, we utilize `chroot` to specify top directory of extracted firmware filesystem as the root, so that the PUT in user-mode emulation can access files with absolute paths correctly.

5.2 Fuzzing

Furthermore, we integrate our enhanced user-mode emulation with AFL by modifying the following strategies.

PUT lifecycle management. In general, AFL uses the main function of the PUT to serve as the fuzzing entry point. It forks a child process at the entry point and conducts the fuzzing in the spawned process. When the PUT executes to the end, AFL exits the spawned process and loops back to the entry point for the next fuzzing iteration. In our system, we further reduce the lifecycle of the fuzzing loop to improve the efficiency. Specifically, we specify the system call that receive the network input as the entry point. Meanwhile, we exit the current fuzzing iteration when we detect that the PUT executes to `poll` or `select` system call that is ready for the new network request as shown in Figure 3.

Entry point of fuzzing. Generally, AFL feeds the test input as a file to the PUT. Unlike AFL, EQUAFL feeds the input to the memory buffer that store the network input. Specifically, we implement the input feeding by instrumenting the system call that receives the network input such as `read`, `recv` and `recvfrom`. After feeding the fuzzed input, we also assign the return value with the length of the fuzzed input.

6 EVALUATION

We implement the prototype of EQUAFL based on QEMU [38] with modifications on both full-system and user-mode code. Besides, we also modify AFL to change the parameters accepted by `afl-fuzz`, so that the arguments and environment variables of the target application could be loaded correctly.

With the prototype, we evaluate the performance of EQUAFL from the following aspects.

- **Compatibility.** We first evaluate EQUAFL to check if it could successfully fuzz the applications which cannot be fuzzed directly by AFL. Also, we compare the system calls sequences/traces of target applications in EQUAFL with full-system emulation-based fuzzing to check the correctness of our enhanced user-mode emulation (§6.2).
- **Efficiency.** We first measure the overhead that was introduced in our system by comparing it with pure user-mode emulation. Then, we evaluate the efficiency of EQUAFL by comparing the throughput with AFL with full-system emulation and Firm-AFL (§6.3).
- **Vulnerability discovery.** We evaluate if EQUAFL can successfully find vulnerabilities in real-world embedded firmware applications (§6.4).

6.1 Experiments Setup

Benchmarks. We use two different datasets as benchmarks in our experiments: ❶ The first dataset contains two sets of standard benchmarks: `nbench` [30] and `lmbench` [31]. ❷ The other dataset consists of 70 embedded firmware images from three major embedded device vendors including D-Link, TRENDnet and NETGEAR. Initially, we collect firmware images by crawling the related vendor's website. Then, we run them on the AFL-Full and Firm-AFL, and obtain the firmware samples that can be successfully emulated and tested. Eventually, We obtain 70 firmware samples in different product series as candidates for the second dataset. The summarized information of firmware images is listed in Table 2.

The programs in the standard dataset are smaller and contain no bugs. So we use them for evaluating the compatibility and efficiency of EQUAFL. The applications we collected from the real-world firmware contain bugs, so we use them to not only evaluate the compatibility and efficiency but also demonstrate how the efficiency boost of EQUAFL can help in finding real-world bugs.

Baselines. We selected three baselines in our experiments.

- AFL-User. AFL-User follows the default AFL setting, where user-mode QEMU is used as the emulator.
- AFL-Full. We integrated AFL with full-system emulation of QEMU to support fuzzing of target applications in full-system emulation. In the system, we utilized the VMI module in DECAF [20] to monitor the process of the target application.
- Firm-AFL. Firm-AFL is a state-of-the-art greybox fuzzer that utilizes both full-system and user-mode emulation to support efficient fuzzing of Linux-based IoT applications.

Configurations. To improve fuzzing efficiency for all the experiments, we use dictionary option in AFL. The keywords used as the dictionary tokens are collected through static analysis of target application. For each target application, we provide a normal network request as the initial seed.

For the compatibility and efficiency evaluation, the experiments are conducted on a 12-core Intel(R) Xeon(R) E5-1650 v3 3.50GHz CPU machine with Ubuntu 18.04.5 LTS operating system and 15.6GB

RAM. For the vulnerability discovery evaluation, the machine is 18-core Intel(R) Xeon(R) CPU E5-2699 v3 2.30GHz CPU machine with Ubuntu 18.04.5 LTS LTS operating system and 188GB RAM.

To mitigate the randomness of the fuzzers, each experiment is repeated for 5 times. In addition, we also followed the suggestions of [24] to conduct Mann-Whitney U tests [33] and calculated the $\hat{A}12$ values [42].

6.2 Compatibility

Table 3 shows the execution results for both EQUAFL and AFL-User on the real-world dataset. The real-world applications are selected following the instructions of Firmadyne. Both AFL-Full and Firm-AFL can successfully emulate all of them. We categorize these results into four states: (1) *CRA* - the application crashes when the fuzzer attempts dry runs with initial seeds; (2) *HAN* - the application hangs when the fuzzer attempts dry runs with the initial seeds; (3) *ERR* - the application process can be initiated by the fuzzer. However, the fuzzer reports some errors such as files not found when it attempts dry runs with the initial seed. (4) *SUCC* - the application process can be initiated by the fuzzer without reporting the errors explicitly.

Table 3: EQUAFL vs. AFL-User execution results

Vendor	NUM	EQUAFL				AFL-User			
		SUCC	ERR	HAN	CRA	SUCC	ERR	HAN	CRA
D-Link	30	28	1	1	0	0	27	3	0
TRENDnet	11	9	0	2	0	0	7	4	0
NETGEAR	29	29	0	0	0	0	2	25	2
SUM	70	66	1	3	0	0	36	32	2

In total, 66 out of 70 applications can reach *SUCC* state in EQUAFL. In contrast, none of the applications can execute in AFL-User correctly with the default user-mode emulation. For D-Link and TRENDnet firmware images, EQUAFL can run most of applications into *SUCC* state, while AFL-User runs most of them into *ERR* state. For NETGEAR firmware images, EQUAFL can run all the applications into *SUCC* state, while AFL-User runs them into other incorrect states. The result shows that EQUAFL is much more effective in successfully executing the real-world applications than AFL-User.

Furthermore, we depict the emulation accuracy of EQUAFL by comparing the execution traces of the same seed with EQUAFL and with AFL-Full. Here, we focus on the correctness of the system call sequences on the execution traces. Note that we only collect the system calls sequence after the input has been received by the application, for two reasons: First, a network application in the full system emulation normally waits for a while until it receives the network input. But the network emulation strategy in EQUAFL is different. EQUAFL handles the network request only when it first encounters a network socket. As a result, the system call sequences before receiving the network input are significantly different in both systems. Second, the fuzzer mainly focuses on the monitoring application execution after the input is received. Thus, we only need to check the correctness of the system call sequences after the input is processed by the application.

We use Levenshtein algorithm [37] to calculate the similarity of system call sequences. Figure 5 shows the sequence similarity results. Among 66 applications in *SUCCESS* state, 44 applications are identical in the system call sequences comparison. Meanwhile, 16 applications have high similarity, which is larger than 89%. Only six applications are totally different in the system calls sequence comparison. With further investigation, we find that the differences in these six applications are mainly because these applications involve inter-process communication which causes the system call sequences to vary. In summary, EQUAFL can execute the target application almost identically to full-system emulation on the real-world dataset.

Last but not least, we also evaluate EQUAFL on the standard dataset to see if it can work well. Specifically, we used nbench to evaluate the accuracy of the enhanced user-mode emulation in EQUAFL. If nbench generates output without errors, it proves that the emulation of EQUAFL is correct. Our evaluation result (see [49]) shows that the enhanced user-mode emulation can execute all the programs in nbench correctly.

In summary, EQUAFL can execute all the programs in the standard dataset and most (66 out of 70) programs in the real-world dataset successfully. Moreover, for the successfully executed programs, their system call sequences are mostly identical to full-system emulation. To conclude, the compatibility of EQUAFL is comparable to full-system emulation and is much better than pure user-mode emulation.

6.3 Efficiency

We evaluate the efficiency of EQUAFL from two aspects: ❶ We measure the fuzzing throughput of EQUAFL and compare it with AFL-Full and Firm-AFL. ❷ We evaluate the overhead of enhanced user-mode emulation in EQUAFL over pure user-mode emulation. Since the real-world applications could not be executed properly with pure user-mode emulation, we use the standard benchmarks to complete overhead evaluation.

Real-world Dataset. We first evaluate the throughput performance of EQUAFL by comparing it with AFL-Full and Firm-AFL. Since the main contribution of EQUAFL is to accelerate the execution of application on the emulator, we did not change any of the fuzzing strategies of AFL. Thus, the fuzzer in EQUAFL is the same as the ones used in Firm-AFL and AFL-Full. As a result, the execution of the target application on the same input can reflect the performance gain of EQUAFL. Therefore, we collect the 1188 seeds from honggfuzz project [18] to evaluate the throughput of EQUAFL.

For each application running in EQUAFL, we first test all the seeds by turns and calculate the average execution time per seed. We repeat the experiment 5 times and obtain the average value ave_t. Then, the average throughput TH_{EQ} is deduced as $\frac{1}{ave\_t}$. Likewise, We obtained the corresponding values TH_{Full} and TH_{Firm} from AFL-Full and Firm-AFL. The result of TH_{EQ}, TH_{Firm} and TH_{Full} for applications that can execute successfully on baselines with seeds is shown in Figure 4. Finally, we worked out the throughput performance improvement $Imp_{Full} = \frac{TH_{EQ}}{TH_{Full}}$ and $Imp_{Firm} = \frac{TH_{EQ}}{THFirm}$ for each application. The results shows TH_{Full} ranges from 5.1

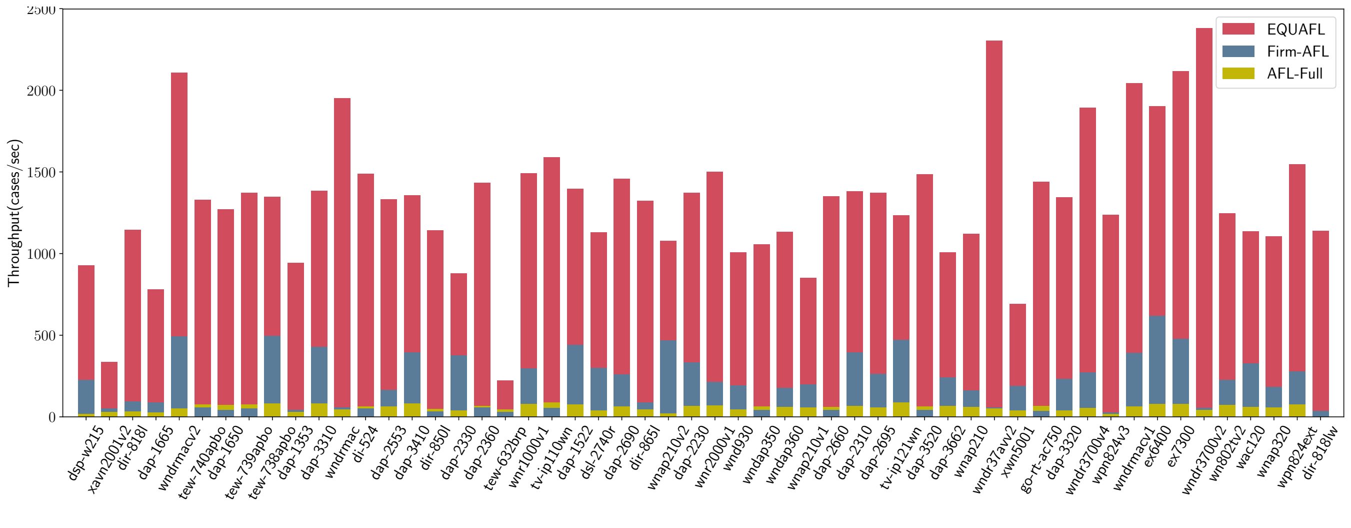


Figure 4: Average execution speed of the evaluated techniques. (Longer is better.)

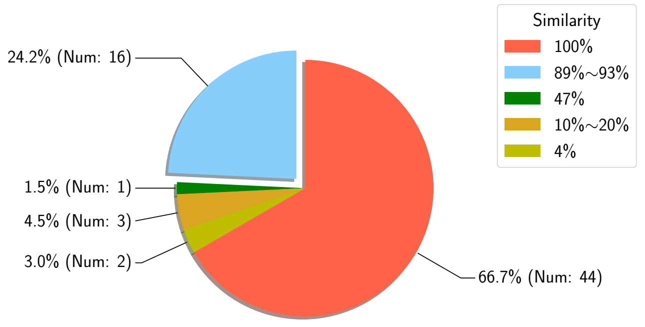


Figure 5: System call sequences similarity distribution

to 78 times (26 times on average), and TH_{Firm} ranges from 2.3 to 48 times (14 times on average), which demonstrates that EQUAFL outperforms state-of-the-art in fuzzing speed.

Standard Dataset. We use both nbench and lmbench to evaluate the overhead of the enhanced user-mode emulation. Specifically, nbench is used to test CPU and memory capabilities of a system. The result (see [49]) shows that the performance of EQUAFL is similar to the performance of pure user-mode emulation because the CPU execution mechanism remains unchanged in the enhanced user-mode emulation. We then use a system call overhead benchmark named lmbench for evaluation. The result (see [49]) shows that the overhead is marginal for both file-related and network-related system calls.

In summary, EQUAFL can execute the real-world applications 26 times faster than AFL-Full and 14 times faster than Firm-AFL on average. Moreover, the overhead of EQUAFL comparing to pure user-mode emulation is shown to be marginal on the standard dataset. This shows that EQUAFL enjoys significantly better efficiency comparing to state-of-the-art techniques.

Table 4: Time to Exposure (TTE) for First Vulnerability (The A12 values are highlighted in the bold if its corresponding *Mann-Whitney U* test is significant.)

Product	EQUAFL	Firm-AFL			AFL-Full		
	μTTE	μTTE	Factor	$\hat{A}12$	μTTE	Factor	$\hat{A}12$
WN2000RPTv1	2220 s	9916 s	4.47	**0.76**	20409 s	9.19	**1.0**
WNDRMACv2	5.0 s	5.2 s	1.04	0.60	5.0 s	1.00	0.50
DIR-825	2011 s	12082 s	6.01	**0.76**	24266 s	12.06	**0.96**
DSP-W215	5 s	5 s	0.96	0.4	39 s	7.37	**1.0**
DSL-2740R	214 s	391 s	1.82	**1.0**	1400 s	6.52	**1.0**
DAP-2330 (vul #1)	42293 s	86400 s	N/A	**1.0**	86400 s	N/A	**1.0**
DAP-2330 (vul #2)	52002 s	86400 s	N/A	**1.0**	86400 s	N/A	**1.0**
DAP-2330 (vul #3)	51972 s	86400 s	N/A	**1.0**	86400 s	N/A	**1.0**
DAP-2330 (vul #4)	80700 s	86400 s	N/A	**1.0**	86400 s	N/A	**1.0**

6.4 Vulnerability Discovery

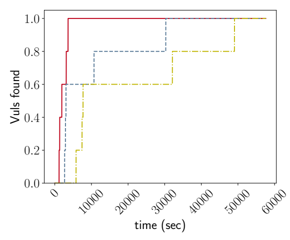
We applied EQUAFL to the real-world applications in an effort of revealing their vulnerabilities. As shown in Table 5, EQUAFL can discover ten vulnerabilities in total, which affect eighteen embedded device series. We then manually analyze the root cause of these vulnerabilities, and finally confirm that nine are NULL pointer dereference vulnerabilities and one is integer overflow vulnerability. We also search vulnerability information online, and only one of the vulnerabilities was revealed by other security researchers before. To our best knowledge, the rest nine vulnerabilities are still unknown to the public. We have reported them to the manufacturers, and six of them have been confirmed as zero-days and assigned CVE numbers until now.

We run 24-hour experiments on the buggy real-world applications to compare the fuzzing result on EQUAFL, Firm-AFL and AFL-Full. First, we select one firmware application for each vulnerability. For firmware images that have multiple vulnerabilities, we still choose one test application. Then, we conduct each fuzzing experiment with five times repeats in parallel for 24 hours.

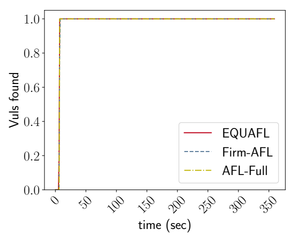
Figure 6 shows the average number of unique vulnerabilities found over time by EQUAFL, Firm-AFL and AFL-Full. The line in

Table 5: Discovered vulnerabilities

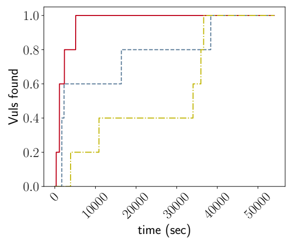
Vendor	Vulnerable Product & Firmware Version	Vulnerable Application	Vulnerability NUM (Unknown)	CVE
NETGEAR	WN2000RPTv1 (1.0.1.20), WPN824EXT (1.1.1), WNR2000v1 (1.1.3.9), WNR1000v1 (1.0.1.5), WPN824v3 (1.0.8)	`/bin/boa`	1 (1)	N/A
NETGEAR	WNDRMACv1 (1.0.0.20), WNDRMACv2 (1.0.0.4), WNDR3700v2 (1.0.0.8), WNDR37AVv2 (1.0.0.10), WNCE4004(1.0.0.22)	`/usr/sbin/uhttpd`	2 (2)	N/A
D-Link	DIR-825 (2.01EU)			
TRENDnet	TEW-632BRP (1.10.31), TEW-634GRU (1.01.06), TEW-652BRP (1.10.14), TEW-673GRU (1.00.36)	`sbin/httpd`	1 (1)	CVE-2021-29296
D-Link	DSP-W215	`/usr/bin/lighttpd`	1 (1)	CVE-2021-29295
D-Link	DSL-2740R (UK_1.01)	`/userfs/bin/boa`	1 (1)	CVE-2021-29294
D-Link	DAP-2330 (1.01)	`/sbin/httpd`	4 (3)	CVE-2021-28838 CVE-2021-28839 CVE-2021-28840



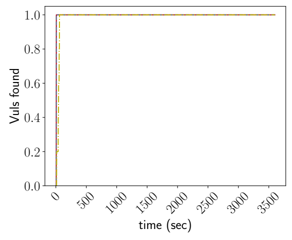
(a) `/bin/boa` (WN2000RPTv1)



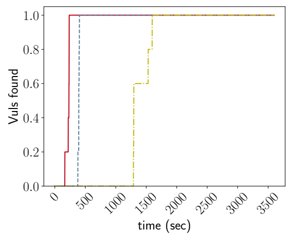
(b) `/usr/sbin/uhttpd` (WNDRMACv2)



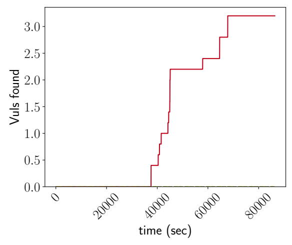
(c) `/sbin/httpd` (DIR-825)



(d) `/usr/bin/lighttpd` (DSP-W215)



(e) `/userfs/bin/boa` (DSL-2740R)



(f) `/sbin/httpd` (DAP-2330)

Figure 6: Unique vulnerabilities found over time.

Figure 6 is the average number of detected unique vulnerabilities. The result shows that EQUAFL can find all vulnerabilities faster than both Firm-AFL and AFL-Full (except for one vulnerability where EQUAFL and Firm-AFL tied). Table 4 shows the time to expose the first vulnerability for each technique. μTTE measures the average time over the five runs. The factor improvement (Factor) measures the performance gain of μTTE compared with the baselines. $\hat{A}12$ means if we randomly pick one run out of the five repeats, by what chance can EQUAFL perform better than the baselines. We can see that EQUAFL can find first vulnerability significantly faster than both AFL-Full and Firm-AFL.

In summary, EQUAFL can detect vulnerabilities much faster than AFL-Full and Firm-AFL. This demonstrates that the enhanced user-mode emulation of EQUAFL indeed helps to boost the vulnerability detection capability of the greybox fuzzer.

7 THREATS TO VALIDITY

The threats to validity come from three aspects. First, EQUAFL can support the emulation of NVRAM peripheral operation in embedded application. Unfortunately, some applications may access other customized hardware peripherals during execution, which cannot be supported. We will provide the emulation of more customized peripherals in future. Second, EQUAFL takes heuristic strategies to bypass the inter-process communication with other applications. However, these coarse-grained heuristics may fail in some cases whose subsequent operations are affected by the communication content. Third, EQUAFL can find vulnerabilities in a single application from embedded firmware. Vulnerabilities that spread across multiple applications cannot be revealed [9, 39].

8 RELATED WORK

We summarize automatic vulnerability discovery techniques for embedded systems from the following perspectives.

Static analysis. Several techniques [14, 15, 46] have been proposed to perform large-scale homology analysis on firmware images to find similar vulnerabilities. However, these techniques need to utilize code features of known vulnerabilities, which is limited in finding unknown vulnerabilities. PIE [13] utilizes machine learning algorithms to identify vulnerable functions in embedded firmware. Firmalice [40] combines static analysis and symbolic execution techniques to discover the authentication bypass vulnerabilities in firmware. DTaint [12] performs fine-grained data flow and structure analysis to find taint-style vulnerabilities in firmware. Karonte [39] and SaTC [9] can efficiently discover vulnerabilities in embedded firmware by utilizing the interaction information from both front-end and back-end programs in embedded firmware. However, these static techniques can only produce the alerts and the vulnerabilities need further manual validation.

Dynamic analysis. Most dynamic analysis techniques are orthogonal to EQUAFL. Avatar [47] provides a hybrid emulation framework for embedded devices, which combines the software emulator and the real device. Firmadyne [5] and FirmAE [23] develop full software emulators for Linux-based embedded firmware images. In our work, we utilize the Firmadyne to support the full-system emulation of Linux-based embedded firmware.

Fuzzing. The majority of existing fuzzing techniques focuses on improving efficiency rather than applicability [3, 4, 6, 7, 10, 11, 16, 26–29, 43–45], which means that theoretically their methods are orthogonal to EQUAFL and can be integrated by EQUAFL. For improving applicability, [1, 22, 48] proposed several methods for creating usable fuzz drivers for testing library targets. For fuzzing software inside embedded systems, IOTFUZZER [8] performs fuzzing by generating effective protocol inputs directly to devices. Muench et al. [32] proposes a fuzzing framework of embedded devices by integrating the boofuzz [36] with diverse emulator approaches. However, they are black-box fuzzing approaches. Firm-AFL [51] combines both full-system and user-mode emulation to perform the fuzzing of applications on Linux-based IoT firmware applications. However, if the application has a large number of system calls that should be forwarded to the full-system emulation, the execution would switch between two emulation modes frequently, which eventually leads to the low efficiency. Firmcorn [19] uses CPU emulator to execute the vulnerable code in firmware which achieves high efficiency of vulnerability-oriented fuzzing. However, register and contextual memory information are still inadequate for accurate firmware code execution. Other techniques such as Firm-Fuzz [41] and EM-Fuzz [17] still adopt the full-system emulation for embedded firmware, which suffer from efficiency problems.

9 CONCLUSION

We proposed an efficient fuzzing framework EQUAFL for network applications in Linux-based embedded firmware. The framework allows to fuzz IoT applications with enhanced user-mode emulation which avoids the cost of full-system emulation of QEMU. We evaluate EQUAFL on standard benchmarks and 70 real-world applications from three major embedded device vendors including D-Link, TRENDnet and NETGEAR. The result shows EQUAFL outperforms AFL in fuzzing compatibility and it outperforms AFL-Full and Firm-AFL in fuzzing efficiency. EQUAFL has already found ten vulnerabilities including six CVEs among these firmware images. In the future, we attempt to support more customized hardware peripherals to further improve the compatibility of EQUAFL.

DATA AVAILABILITY STATEMENT

We have uploaded the artifact to Zenodo [50], which includes the source code and README.md to guide users to run the docker container and perform the testing.

ACKNOWLEDGEMENT

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from the industry partner(s), the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 975–985. https://doi.org/10.1145/3338906.3340456

[2] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506. https://doi.org/10.1109/tse.2017.2785841

[5] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Network and Distributed System Security Symposium, NDSS*. https://doi.org/10.14722/ndss.2016.23415

[6] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. 2325–2342.

[7] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108. https://doi.org/10.1145/3243734.3243849

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Networked and Distributed System Security Symposium (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23159

[9] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. 2021. Sharing More and Checking Less: Leveraging Common Input Keywords to Detect Bugs in Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 303–319.

[10] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725. https://doi.org/10.1109/sp.2018.00046

[11] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596. https://doi.org/10.1109/sp40000.2020.00002

[12] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. 2018. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 430–441. https://doi.org/10.1109/DSN.2018.00052

[13] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. 2015. PIE: Parser identification in embedded systems. In *Annual Computer Security Applications Conference (ACSAC'15)*. https://doi.org/10.1145/2818000.2818035

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*.

[15] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 480–491. https://doi.org/10.1145/2976749.2978370

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696. https://doi.org/10.1109/sp.2018.00040

[17] Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. 2020. EM-Fuzz: Augmented Firmware Fuzzing via Memory Checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (Nov. 2020), 3420–3432. https://doi.org/10.1109/TCAD.2020.3013046

[18] Google. 2022. Honggfuzz: security oriented software fuzzer. Supports evolutionary, feedback-driven fuzzing based on code coverage (SW and HW based). https://github.com/google/honggfuzz.

[19] Zhijie Gui, Hui Shu, Fei Kang, and Xiaobing Xiong. 2020. FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution. *IEEE Access* 8 (2020), 29826–29841. https://doi.org/10.1109/access.2020.2973043

[20] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, 248–258. https://doi.org/10.1145/2610384.2610407

[21] Hex-Rays. 2022. IDA Pro. https://www.hex-rays.com/products/ida/.

[22] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*. 2271–2287.

[23] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Annual Computer Security Applications Conference (ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 733–745. https://doi.org/10.1145/3427228.3427294

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3243734.3243804

[25] ReFirm Labs. 2021. Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk.

[26] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ASE '18*. ACM, 475–485. https://doi.org/10.1145/3238147.3238176

[27] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637. https://doi.org/10.1145/3106237.3106295

[28] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544. https://doi.org/10.1145/3338906.3338975

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[30] Uwe F. Mayer. 2017. nbench. https://www.math.utah.edu/~mayer/linux/bmark.html.

[31] Larry McVoy and Carl Staelin. 2012. LMbench - Tools for Performance Analysis. http://www.bitmover.com/lmbench/.

[32] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Network and Distributed System Security Symposium (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23166

[33] Nadim Nachar. 2008. The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology* (2008). https://doi.org/10.20982/tqmp.04.1.p013

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

[35] Krebs on Security. 2016. Source Code for IoT Botnet 'Mirai' Released. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released.

[36] Joshua Pereyda. 2016. boofuzz. https://github.com/jtpereyda/boofuzz.

[37] Python. 2021. python-Levenshtein 0.12.2. https://pypi.org/project/python-Levenshtein/.

[38] QEMU. 2022. Official QEMU mirror. https://github.com/qemu/qemu.

[39] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware. In *In Proceedings of the IEEE Symposium on Security & Privacy (S&P)*. https://doi.org/10.1109/sp40000.2020.00036

[40] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Network and Distributed System Security Symposium (NDSS'15)*. https://doi.org/10.14722/ndss.2015.23294

[41] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. 2019. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P'19)*. Association for Computing Machinery, 15–21. https://doi.org/10.1145/3338507.3358616

[42] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101

[43] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 999–1010. https://doi.org/10.1145/3377811.3380386

[44] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 579–594. https://doi.org/10.1109/sp.2017.23

[45] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777. https://doi.org/10.1145/3377811.3380396

[46] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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)*. https://doi.org/10.1145/3133956.3134018

[47] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS'14)*. https://doi.org/10.14722/ndss.2014.23229

[48] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-source SDK Libraries. In *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.

[49] Yaowen Zheng. 2022. EQUAFL: Efficient Greybox Fuzzing of Network Applications in Embedded Linux Firmware via Enhanced User-mode Emulation. https://sites.google.com/view/equafl/.

[50] Yaowen Zheng. 2022. EQUAFL_artifact. https://doi.org/10.5281/zenodo.6580348.

[51] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1099–1114.