FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis

Mingeun Kim
The Affiliated Institute of ETRI
rla5072@nsr.re.kr

Dongkwan Kim
KAIST
dkay@kaist.ac.kr

Eunsoo Kim
KAIST
hahah@kaist.ac.kr

Suryeon Kim
Ministry of National Defense
c16192@kaist.ac.kr

Yeongjin Jang
Oregon State University
yeongjin.jang@oregonstate.edu

Yongdae Kim
KAIST
yongdaek@kaist.ac.kr

ABSTRACT

One approach to assess the security of embedded IoT devices is applying dynamic analysis such as fuzz testing to their firmware in scale. To this end, existing approaches aim to provide an emulation environment that mimics the behavior of real hardware/peripherals. Nonetheless, in practice, such approaches can emulate only a small fraction of firmware images. For example, Firmadyne, a state-of-the-art tool, can only run 183 (16.28%) of 1,124 wireless router/IP-camera images that we collected from the top eight manufacturers. Such a low emulation success rate is caused by discrepancy in the real and emulated firmware execution environment.

In this study, we analyzed the emulation failure cases in a large-scale dataset to figure out the causes of the low emulation rate. We found that widespread failure cases often avoided by simple heuristics despite having different root causes, significantly increasing the emulation success rate. Based on these findings, we propose a technique, arbitrated emulation, and we systematize several heuristics as arbitration techniques to address these failures. Our automated prototype, FirmAE, successfully ran 892 (79.36%) of 1,124 firmware images, including web servers, which is significantly (≈4.8x) more images than that run by Firmadyne. Finally, by applying dynamic testing techniques on the emulated images, FirmAE could check 320 known vulnerabilities (306 more than Firmadyne), and also find 12 new 0-days in 23 devices.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; • **Computer systems organization → Firmware**.

KEYWORDS

Firmware, embedded device, emulation, dynamic analysis

ACM Reference Format:
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 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3427228.3427294

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ACSAC 2020, December 7–11, 2020, Austin, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427294

1 INTRODUCTION

The number of active Internet of things (IoT) devices is projected to reach 34.2 billion by 2025 [36]. As numerous IoT devices are connected to the Internet [33], they are exposed to cyber threats. For instance, Linux-based IoT devices such as wireless routers and IP cameras are often targeted for a large scale attacks. In the wild, multiple backdoors were discovered from those devices [30, 38], and malwares, such as Mirai and Satori, infected over millions of such devices [5, 31, 32, 37].

To address security problems in such numerous IoT devices, researchers have been focusing on analyzing firmware of those devices in scale. Specifically, a stream of studies take an approach to run device firmware in an emulated environment with virtual hardware and then apply dynamic analysis to the firmware [12, 17, 23–25, 57, 60, 64]. With this approach, one can not only analyze firmware dynamically without obtaining hardware but also utilize cloud infrastructure to scale the security analysis. Among many, Firmadyne [17] is the current state-of-the-art firmware emulation framework that aims at enabling large-scale emulation for IoT devices in general by providing a full-system emulation environment.

Problem: discrepancy in the real and virtual environment. The approach is not a silver bullet in practice because running firmware in a full-system emulation environment often fails due to the inconsistencies between the real and the virtual, emulated environment. Any discrepancies in the emulated environment may lead the firmware execution to an unexpected state, resulting in a failure of emulation and dynamic security analysis. Resolving this emulation discrepancy is challenging because such inconsistencies stem from the wide diversity in IoT device hardware and configurations. Particularly, each of IoT devices equipped with specific hardware devices from a plethora of manufacturers. Moreover, firmware often relies on the configuration vectors, such as data in NVRAM, and an emulated environment may miss such data because the data is only available in hardware. Such convoluted circumstances does not match with the emulation environment of Firmadyne. Its emulator, QEMU [6], only supports few general devices and configurations, and without putting extensive efforts on emulating each device and configuration the problem will never disappear.

To see the effect of this problem in practice, we have obtained 1,124 wireless router and IP camera firmware images from top eight vendors and ran them with Firmadyne. The result is alarming as it can only emulate 183 of them (see Table 1). The majority portion of firmware images (83.72%) left without analysis. Such

a low emulation success rate implies that, although Firmadyne is designed to be generic by providing a full-system emulation environment of a firmware, such an approach does not work in practice, requiring many manual efforts to resolve inconsistencies in emulated environment.

Motivating examples. Next, we show how we can handle the inconsistencies manually as motivating examples. First, we ran the firmware of `D-Link DIR-505L` to test CVE-2014-3936 [16] using Firmadyne. Because the vulnerability is a stack-based buffer overflow in a web service running on firmware, exploitation requires sending HTTP requests via the network interface of the emulated environment. However, when we ran the firmware on Firmadyne, we could not connect to the web service although the web server is running correctly. From our analysis, we figured out that the network configuration in the firmware does not match to the emulated environment, and after we force to configure the network, we were able to trigger the vulnerability. Second, we ran the firmware of `NETGEAR R6250` to test CVE-2017-5521 using Firmadyne. In this case, the emulation failed with a kernel panic in the booting procedure. After we slightly modified the booting and kernel-related configuration to match the virtual environment, we were able to run the firmware and trigger the vulnerability.

Observations and goal. From these two examples, we observed that a slight change in a configuration or device settings, which is easy to apply, may let firmware emulation run without suffering emulation discrepancy problem, which is difficult to handle. In this regard, we believe that Firmadyne *misses many chances of emulating and analyzing IoT firmware images not because of fundamental problems in emulation but because of device setup failures, although these can be easily handled.* To address this issue, we aim at systematizing such heuristics via analyzing many emulation failure cases, and ultimately, we aim to increase the chance of successful firmware emulation than Firmadyne.

Our approach. We achieve this goal by investigating many emulation failure cases as our first step. For the investigation, we collected 1124 firmware images from the top eight vendors [59]: 1079 wireless routers and 45 IP cameras. For the emulation, we specifically focus on emulating web services of wireless routers and IP cameras. This is because the web interface is the part where remote attackers can interact with, and numerous critical vulnerabilities have been found in these services [5, 7, 12, 32, 51]. By using Firmadyne, we investigated 437 emulation failure cases (among 527 firmware images in `AnalysisSet`) and found that most cases fall into the following five categories of problems: 1) boot-related problems, such as an incorrect boot sequence or absence of files, 2) network-related problems, such as mismatches of network interface or improper configuration, 3) non-volatile RAM (NVRAM)-related problems, such as missing library functions or customized formats, 4) kernel-related problems, such as unsupported hardware or functions, and 5) minor problems, such as unsupported commands or timing issues.

Our investigation resulted that *failure cases in each category can be resolved by applying simple heuristics even though they originate from different root causes.* For example, 227 images failed to set up their network interfaces even though their web servers were correctly running. Although the root causes of the failures may vary, such as discrepancy in the number of available network ports, the name of network device, etc., a heuristic that forces setting up the network configuration that works in an emulated environment can resolve the issue and enable dynamic analysis.

Based on this observation, we systemize those heuristics as a technique, coined as **arbitrated emulation**, and develop several arbitration techniques to bypass the failure cases. Instead of strictly following the execution behavior of the firmware as is, arbitrated emulation arbitrates between following the original behavior or injecting proper interventions, i.e., intentional operations. Thereby, it may slightly alter the original behavior of the firmware. However, our goal is not to build an environment identical to the physical device, but to create an environment conducive to the dynamic analysis. In fact, our approach can emulate numerous firmware images that previous approaches failed to emulate, and effectively aid in finding real vulnerabilities.

After designing several arbitrations, we automate and parallelize the entire firmware emulation procedure. Within 4h of testing 1,124 firmware images, our prototype, FirmAE, successfully emulated 892 (79.36% of total) images, which is more than four-times more than Firmadyne (Table 1). Then, we ran exploits of previously known vulnerabilities on the emulated images to verify whether arbitrated emulation is useful for dynamic analysis. As a result, 320 known vulnerabilities were successfully emulated on FirmAE which is 306 more successful cases than Firmadyne. We also built a simple fuzzer on FirmAE, and found 23 unique vulnerabilities in 95 latest devices, and responsibly reported them to the vendors.

In summary, the contributions of our study are as follows:

- We empirically investigate 437 firmware emulation failure cases and systematize failure handling heuristics.
- We propose arbitrated emulation to apply those heuristics to emulation environment. Our prototype, FirmAE, presents a far higher emulation success rate (892 *vs.* 183) than the state-of-the-art framework, Firmadyne.
- We confirm that arbitrated emulation is effective by rediscovering 306 more known vulnerabilities than Firmadyne. Additionally, with a simple fuzzer, FirmAE can find 23 new vulnerabilities over 95 latest devices, out of which 12 were 0-days.
- We release the source code to encourage future studies.[1]

2 BACKGROUND

In this section, we explain how embedded devices are analyzed by citing previous studies and present the state-of-the-art tool that we employed as the basis of our approach.

2.1 Embedded device analysis process

To analyze an embedded device, the target firmware can be obtained and analyzed with/without a physical device.

Firmware collection and unpacking. Typically, firmware can be acquired from vendors' websites, ftp servers, or third-party archives. This can be done manually or by using a web crawler such as Spider [41]. Firmware can also be directly dumped from the flash memories in devices [46], although this requires a physical device.

[1] http://github.com/pr0v3rbs/FirmAE

A firmware image is then unpacked for later analysis. A single image can include multiple contents. For example, Linux-based firmware may have a bootloader, kernel, and filesystem. This image is often compressed in various ways, such as LZMA, ZIP, or Gzip, to save storage. To unpack an image, tools such as Binwalk [26], Firmware-Mod-Kit [27], or FRAK [13] are often employed. In a given image, these tools scan pre-defined signatures of various file headers. When a signature matches, they extract the file from the image, and continue to scan it to the end. Encrypted or customized images also exist, for which signature matching cannot be used; analyzing them is out of the scope of this study.

Analysis with physical devices. The unpacked firmware can be analyzed with real devices. Zaddach *et al.* [62] and Marius *et al.* [44] relayed process execution and peripheral access to real devices and partially emulated target code using a JTAG interface. Similarly, Kammerstetter *et al.* [28, 29] developed a proxy environment using real devices and forwarded character device access to them. Cui *et al.* [14, 15] and Kumar *et al.* [33] conducted a quantitative study of embedded devices connected to the public Internet.

Analysis w/o devices. Another stream of studies have focused on analyzing firmware without physical devices to scale up the analysis. Researchers adopted static approaches on firmware [11, 52]; however, they often produce numerous false positives due to the absence of runtime information. Nevertheless, Costin *et al.* [11] showed statistics of vulnerable devices that have easily crackable passwords or backdoor strings. Shoshitaishvili *et al.* [52] found authentication bypass vulnerabilities using symbolic execution.

In contrast, dynamic analysis can identify vulnerabilities without false positives as it runs the target program directly. However, performing dynamic analysis is not a simple task, as the device firmware has to be emulated. Recent studies [12, 17, 23–25, 57, 60, 64] focused on firmware emulation to overcome the difficulty in obtaining the real hardware, and we further describe these studies in details in the following subsection (§2.2).

2.2 Emulation-based analysis

Firmware emulation has attracted attention, as it does not require real devices and provides useful interfaces for dynamic analysis. The system where the emulation takes place is denoted as the *host* system, and the emulated system is referred to as the *guest* system. Typically, there are two levels of emulation: user- and system-level.

User-level emulation. User-level emulation only emulates the target program inside the firmware and makes the best use of the host system. An example is emulating a web interface. A web interface is a representative service in embedded devices for device administration or maintenance. It serves multiple static contents, such as HTML, or dynamic contents generated by CGI programs. Although static contents can be served with the host environment, dynamic contents may not. This is because they may collide with the host system or depend on custom libraries and device drivers that do not exist in the host system.

System-level emulation. System-level emulation fully emulates the guest system, including the kernel. Because it provides an individual execution environment, various features in kernel and device drivers can be emulated as well. Nevertheless, firmware emulation is extremely difficult, as vendor-specific hardware issues or memory-mapped peripherals should be considered. Without handling them, programs in the emulated firmware often crash.

Consequently, studies have recently struggled to address these issues [12, 17, 23, 25, 57], by creating an emulation environment as similar as possible to the real device. Popular emulators, such as QEMU [6], have been supporting more hardware types, including their peripherals. Costin *et al.* [12] presented a scalable dynamic analysis framework along with several case-studies on various embedded web interfaces. Chen *et al.* [17] emulated non-volatile RAM (NVRAM), which stores various configuration values for programs in the emulated firmware. Gustafson *et al.* [25] modeled memory-mapped I/O (MMIO) operations in peripheral communication. Feng *et al.* [23] attempted to resolve the same issue with machine learning. Recently, Clements *et al.* [10] proposed decoupling the hardware from the firmware.

Analysis. After emulation, vulnerabilities can be checked by using a previously known PoC code [17] or a fuzzer [24, 60, 64]. TriforceAFL [24] is a popular fuzzer targeting a QEMU image, leveraging the American fuzzy lop (AFL) [63]. It is also adopted by Hu *et al.* [60]. In their follow-up study, Zheng *et al.* [64] proposed an optimized emulation approach for dynamic analysis, which switches the context between system- and user-level emulation.

2.3 Challenges in firmware emulation

Emulation-based analysis is advantageous; however, there are numerous challenges when emulating firmware images from diverse vendors, which stem from the non-standardized development process and the discrepancy between the emulated and physical environments. For example, libraries, device drivers, and even kernels in devices differ across vendors; unless these are properly emulated, internal programs cannot be executed.

Devices that access hardware interfaces, such as LED sensors or cameras, have more diversity, as noted in previous studies [23, 25]. Communication between the main device and its peripherals often utilizes memory-mapped IO (MMIO) operations, with pre-defined memory addresses. However, the range of such addresses differs significantly across devices. Consequently, it is difficult to scale this approach to various devices. Chen *et al.* [17] attempted to emulate one such hardware, namely NVRAM, on a large scale. Muench *et al.* [45] underlined device-specific challenges when conducting a dynamic analysis to identify memory corruption vulnerabilities.

Addressing these challenges may be infeasible, unless functions are implemented perfectly as in physical devices. Nevertheless, investigating emulation failure cases and resolving identified issues helps gradually increase the emulation rate, and enable dynamic analysis to improve the security of IoT ecosystem. Therefore, we adopt the state-of-the-art emulation framework, Firmadyne [17], and investigate the failure cases.

2.4 Firmadyne framework

Firmadyne [17] is a state-of-the-art firmware emulation framework, originally designed for a large-scale analysis. Numerous studies [24, 60, 64] have adopted it for dynamic analysis. We also utilized Firmadyne for failure investigation.

After unpacking a firmware image, Firmadyne emulates it with a customized Linux kernel and libraries, which are pre-built to

support various hardware features such as NVRAM. For emulation, Firmadyne emulates the target image twice: the first emulation logs useful information, whereas the second utilizes the logged information. Thus, the customized kernel includes a driver that hooks major system calls to record useful information. For example, they hook `inet_ioctl()` and `inet_bind()` to obtain the name and IP address of the network interface used in the emulated firmware. The custom libraries of Firmadyne also address hardware issues. For example, a library, `libnvram`, stores and returns NVRAM values based on the hard-coded default values.

Although Firmadyne is promising, its emulation rate of network reachability and web service availability is considerably low at 29.4% and 16.3%, respectively. To this end, we carefully investigate the failure cases and propose a technique to address them.

3 DESIGN

3.1 Goal and scope

Goal. Our goal is to successfully emulate the firmware image of embedded devices, specifically running their web services because the web interface of such devices is a critical target for remote attackers. [5, 12, 17, 32, 60, 64]. We do not aim to resolve all the discrepancies in emulated environment. Instead, we aim at a concise emulation for dynamic testing, and our emulation goal can be illustrated with the following properties: 1) booting without any kernel panic, 2) network reachability from the host, and 3) web service availability for dynamic analysis. We are aiming at holding these properties as they are the minimum requirements for running web services without suffering issues in firmware emulation. Thus, we check the emulation success rate by checking the network reachability and web service availability of the target firmware.

Scope. Among various embedded devices, we select wireless routers and IP cameras as our analysis targets because of their presence in our daily lives and as they often become attack targets. In fact, many botnets [5, 32] target them to launch large-scale DDoS attacks. Note that other embedded devices that share similar characteristics can be addressed with our approach as well.

3.2 Arbitrated emulation

To achieve this goal, we propose a technique, which we refer to as *arbitrated emulation*. Whereas previous approaches [12, 17, 23, 25, 57] have striven to ensure that the target firmware operates alike the physical device, which is a difficult goal, arbitrated emulation does not completely follow the original execution procedure of the target firmware. The key idea behind arbitrated emulation is that ensuring high-level behavior is sufficient to perform dynamic analysis on internal programs, which is relatively easy to do, rather than finding and fixing the exact root causes of emulation failures. The high-level behavior mentioned here can be readily modeled by skilled analysts based on their target and emulation goal. In this study, we use the model defined in §3.1.

One key feature of arbitrated emulation is that it employs *intervention*. The intervention indicates an intentionally added action, which may differ from the behavior of the physical device. This action makes it possible to bypass unaddressed issues assuming that they do not strongly influence the behavior of the target program inside the emulated firmware. The procedure that arbitrates

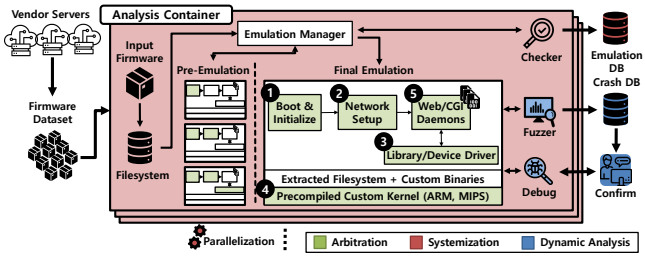

Figure 1: FirmAE architecture overview

between following the firmware as is and applying an intervention is called *arbitration*. An intervention can be implemented in various ways, as needed, and it can be injected into the appropriate steps of the emulation procedure, namely the *arbitration point*. Proper arbitration points can be noted by analyzing violation cases of the given high-level behavioral model. Then, interventions are injected in these arbitration points. As interventions focus on high-level behaviors, those obtained from a small set of firmware images can be widely applied to other firmware images that suffer from similar failure cases, even though they have different root causes.

Our interventions take advantages of the abstract design of Linux-based firmware. We conducted a preliminary study on our dataset and discovered that appropriate interventions can aid the emulator to bypass numerous unsolved issues. For example, when a network setting procedure is stopped due to an unknown peripheral access or insufficient NVRAM support, an intervention that forces the configuration of a fixed network setting can resolve the issue regardless of the root cause. Although the arbitrated emulation may violate the main concept of the full-system emulation, we hypothesized that small discrepancies introduced by interventions only have a slight effect on the behavior of the target program. In fact, we support this hypothesis by successfully running emulated web services in 892 firmware images from 1,124 images, and we found 12 0-day vulnerabilities by conducting dynamic security analysis.

3.3 FirmAE

We implemented our prototype of arbitrated emulation, FirmAE, based on Firmadyne [17]. The overall architecture of FirmAE is illustrated in Figure 1. FirmAE emulates a firmware image similar to Firmadyne on a pre-built customized Linux kernel and libraries, as described in §2.4. It also emulates a target image twice to collect various system logs and utilize the information for further emulation. We refer to the former emulation step as pre-emulation and the latter as final emulation. The arbitrations applied in FirmAE can be categorized into five, which are derived by our failure case investigation on `AnalysisSet`. We describe the details of each arbitration in §4, and compare the emulation results with those of Firmadyne in §5.1. We built additional interfaces for dynamic analysis on FirmAE (§5.3), and the analysis results is described in §5.4.

Automation. For a large-scale analysis, FirmAE needs to be fully automated. Naturally, numerous steps of Firmadyne are automated; however, it still requires some user interaction. For example, users must first extract the filesystem of the target firmware with specific options. Then, they evaluate whether the filesystem is successfully extracted and retrieve the architecture information. Subsequently, they make a firmware image for QEMU and collect information

in pre-emulation. Finally, they run a script for final emulation and perform dynamic analysis. We automated all such interactions and added an automated evaluation procedure for network reachability and web service availability. For this, we built a module in FirmAE that periodically runs `ping` and `curl` commands.

Parallelization. We also parallelized emulation to effectively evaluate numerous firmware images, leveraging containerization with Docker [40]. Each firmware image is emulated independently in each container, which is equipped with all required packages and dependencies. This enables the quick and reliable emulation of a target image. FirmAE emulates firmware in parallel, by running multiple container instances.

With containerization, we can take advantages of abstracting the network connection between the host and the guest systems. QEMU [6], which FirmAE utilizes for the emulation, creates an additional network interface, *TAP*, in the host system. This interface is linked to one of the guest network interfaces. Thus, each emulated firmware should have an independent TAP interface with a unique IP address in the host system, otherwise a network collision will take place. Containerization isolates the network environment of each container. Consequently, packets from the host system can be properly routed to the guest even in parallel emulation. We also place the checker and analysis engine inside each container.

3.4 Experimental setup

Dataset. Our dataset comprises the top eight vendors in the wireless home router market [59]. We collected 1306 firmware images from the vendors' websites and extracted filesystems from the collected images by unpacking them with Binwalk [26], as described in §2.1. Then, we filtered them by verifying whether the operating system of each image has one of the three architectures: ARM little endian (ARMel), MIPS little endian (MIPSel), and MIPS big endian (MIPSeb). These architectures occupy more than 97% of our initial collection. We prepared IP camera firmware in the same manner.

Our final dataset includes total 1124 firmware images, 1079 of which are wireless routers, and 45 are IP cameras. We divide them into three datasets: `AnalysisSet`, `LatestSet`, and `CamSet`. Their brief summary is presented in Table 1 along with the emulation result, and its detailed version is shown in Table 4 in Appendix. The `AnalysisSet` consists of 526 outdated images from 3 vendors, whereas the `LatestSet` and `CamSet` only contain the latest firmware images as of Dec. 2018. The `LatestSet` has 553 latest images from 8 vendors including the vendors covered by the `AnalysisSet`, and the `CamSet` includes 45 latest images from 3 vendors. Accordingly, the `AnalysisSet` may include multiple firmware versions per device, whereas the `LatestSet` and `CamSet` have only one image per device. There is no intersection among the datasets, i.e., they do not share any identical image. We used the `AnalysisSet` to analyze emulation failure cases. By analyzing them, we found several arbitration points that can help increase the emulation rate (§4). We applied those arbitrations in FirmAE, and evaluated it with the `LatestSet` and `CamSet` (§5).

Environment. All our experiments were conducted on a server equipped with four Intel Xeon E7-8867v4 2.40 GHz CPUs, 896 GB DDR4 RAM, and 4 TB SSD. We installed Ubuntu 16.04 with PostgreSQL v9.5.14 [42] and Docker v18.09.4 [40] on the server.

4 ARBITRATION OF FAILURE CASES

The key of arbitrated emulation is to depict arbitration points that can help the emulator to bypass the failure. Therefore, we first analyzed the failure cases on `AnalysisSet` based on the high-level behavioral model. For a large-scale analysis, we applied FirmAE's automation and parallelization without any arbitration, such that the emulation part is the same as that of Firmadyne. Notably, web servers of only 16.9% images were emulated (§5.1). For a neat explanation, we categorized the failure cases by their arbitration points: boot (§4.1), network (§4.2), NVRAM (§4.3), kernel (§4.4), and others (§4.5). In this section, we explain them in details.

Note. Identifying arbitration points and devising appropriate interventions require empirical investigation, and we believe that our study can contribute to future research in this field.

4.1 Boot arbitrations

We encountered the first issues at the early stage of the booting procedure, which made the emulation fail with kernel panic.

Improper booting sequence. The main cause of an improper booting sequence is that the program used for system initialization is not properly executed. Generally, most systems require initialization in the booting procedure. In the Linux kernel, initialization is often performed by a program called `init`, and the kernel attempts to find this program by checking pre-defined paths, such as `/sbin/init`, `/etc/init`, and `/bin/init`. However, some firmware images have customized paths for initializing programs, such that the kernel fails to execute the programs and crashes.

This failure often happens in NETGEAR firmware images. After analyzing them, we found that they use the name `preinit`, which is often used by an open-source embedded device project OpenWrt [22], and we verified that they are indeed implemented upon it. We also found that some TP-Link images utilize `preinit` as well. To address this problem, Firmadyne built a script that searches and executes a hard-coded list of files frequently accessed for initializing programs. However, these candidates are not sufficient to account for the diverse paths of initializing programs in the wild.

We propose another approach that utilizes information from the kernel of the target firmware. Specifically, we created an intervention at the beginning of the booting process, which extracts useful information in the kernel of the image. Specifically, we utilize a kernel's command line string, which is used for default configuration of the kernel in the booting procedure. Note that such a string is pre-defined in the development stage, so it is naturally embedded in the kernel image. This information may include an initializing program path, console type, root directory, root filesystem type, or memory size. For example, from one kernel image in NETGEAR firmware, we could obtain a string of `console=ttyS0,115200 root=31:08 rootfstype=squashfs init=/etc/preinit`. We can recognize the initializing program path of `/etc/preinit`, console type of `ttyS0` having the 115200 baud rate, and root filesystem type of `squashfs`. By configuring the emulated environment with the information obtained from the original kernel, the guest system could be properly initialized without failure, even if initializing programs have unusual paths. If we fail to extract any information, we find the initialization program such as `preinit` or `preinitMT` from the extracted filesystem.

Missing filesystem structure. Other failure cases occur due to the absence of files or directories. When internal programs access such paths, they crash, and the emulation stops. Firmadyne attempted to address this by creating and mounting hard-coded paths such as `proc`, `dev`, `sys`, or `root` at the beginning of the custom booting script. Some hard-coded paths certainly worked; for example, making `/etc/TZ` or `/etc/hosts` helped several cases of this failure. However, this approach cannot account for diverse cases. Furthermore, as it forcibly creates files and directories before the firmware initializes itself, it collides with internal programs, which create and mount other files or directories in the same paths.

We arbitrated this by inserting an intervention, which is similar to that of the previous case, but retrieves information from a filesystem rather than a kernel. Before emulating a given image, we extracted all strings from executable binaries in its filesystem. Then, we filtered them to obtain strings that are highly likely to indicate paths and prepared the file structure based on the paths. In particular, we chose strings that start with general Unix paths, such as `/var`, or `/etc`.

4.2 Network arbitrations

After completion of the booting procedure, the network should be configured such that the host system can communicate with the guest system, and eventually dynamic analysis can be performed. For network communication, QEMU requires the host to create an additional network interface, *TAP*. This TAP interface is connected to a network interface in the guest system. Then, the host and guest communicate through it.

However, properly configuring a TAP interface is not trivial, as it should be set up with specific options that correspond to the type of the target network interface. This network interface type could be Ethernet, wireless LAN (WLAN), network bridge, or virtual LAN (VLAN). As statically distinguishing the interface type in the guest system is not easy, a target image needs to be emulated once.

Firmadyne emulates a given image twice (§2.4). In the first emulation, namely pre-emulation, Firmadyne collects kernel logs by hooking the system calls. Since the collected logs include the names and IP addresses of the network interfaces accessed during the emulation, they could be utilized for network configuration in the final emulation. Nevertheless, numerous images still suffered failure.

Invalid IP alias handling. Assigning multiple IP addresses to a single network interface is termed IP aliasing [58]. It is prevalent in routers, as it enables the management of services separately by IP address. In IP aliasing, a network interface makes multiple instances of itself, and each instance is assigned a unique IP address. For example, a bridge interface, `br0`, with an IP address of `192.168.1.1`, can have IP aliases of `169.254.39.3` and `1.1.1.1`, which are assigned to its instances `br0:0` and `br0:1`, respectively. Then, `br0` is linked to an Ethernet interface, `eth0`. Here, `br0` can be accessed with any of these IP addresses.

Failure cases related to this IP aliasing are often found in D-Link images. After investigating them, we found that they are caused by the fact that Firmadyne does not properly handle IP aliasing. The problem occurs during the Firmadyne network configuration in the host system. At the pre-emulation step, IP aliases are logged by the kernel. Then, Firmadyne parses the log and tries to assign all the logged IP addresses to a corresponding interface in the guest. Then, it adds static routing rules for those IP addresses to link them to a TAP interface in the host. Here, multiple routing rules are added to a single TAP interface, which makes the network collide.

With the knowledge of IP aliasing, FirmAE arbitrates this by letting the host system use its default routing rule. In particular, even though IP aliasing is used, once the guest's network interfaces are linked to the host's TAP interfaces, all packets are automatically routed between the host and guest. Thus, these cases require no intervention, demonstrating the importance of placing an intervention for the right situation.

No network information. Some firmware images do not contain any information on connectable network interfaces, such as `eth`, in their kernel logs. Those images only configure the loopback interface (`lo`) without setting other network interfaces. Due to the lack of connectable network interface, these images cannot be accessed from the host system. Moreover, some images attempt to bind their web servers to a network interface, which does not exist, and consequently crash.

After analyzing the cases, we found that some images use the dynamic host configuration protocol (DHCP) to retrieve IP addresses from a DHCP server, for their WAN interface. The DHCP is a popular protocol to set up a network interface in endpoint devices, as it does not require any user interaction. In general, wireless routers act as DHCP servers themselves to assign IP addresses to their LAN interfaces to which their clients are connected. However, they can also retrieve an IP address from external DHCP servers to connect their WAN interface to the Internet, unless a user manually configured it. Indeed, our analyzed images attempted to retrieve an IP address with the DHCP through the connection between their WAN interface and the TAP interface of the host system. However, as a DHCP server is not present in the emulated environment, the emulated firmware fails to obtain an IP address and configure a network interface. Furthermore, as no network interface is configured, a bridge interface, which groups multiple network interfaces, could not be arranged. Consequently, internal programs that are bound to these network interfaces cannot run properly.

We first attempted to address this with QEMU's internal DHCP server, such that guest's network interfaces can retrieve IP addresses from the server. However, some images still do not have network interfaces, even after setting the DHCP server. This may arise from insufficient support of peripherals. If any program during the network configuration accesses such peripherals, it crashes or acts abnormally and eventually fails to configure the network.

FirmAE arbitrates these cases with an intervention that forcibly configures the network with a default setting. Specifically, we set an Ethernet interface, `eth0`, with an IP address of `192.168.0.1`. After the Ethernet interface is set up, it is linked with a default bridge interface, `br0`, for those images whose kernel log contains bridge interface information. This simple intervention significantly helps emulate web services (§5.1).

Multiple network interfaces in ARM. To support multiple network interfaces, an appropriate machine, onto which the target firmware will be loaded, must be chosen. We selected `virt`, one of the machines supported by QEMU, by following the approach employed in a previous study [17]. This performs well for several

firmware images; however, it fails to emulate ARM firmware images with multiple network interfaces. Firmadyne attempts to address this multi-interface problem by preparing a fixed number (four) of dummy interfaces. Its basic assumption is that the number of interfaces should be more than or equal to the suffix of the interface name, which is extracted from kernel logs. For example, if `eth1` is logged, it is highly likely that `eth0` exists as well. However, almost all ARM images are still not emulated.

We carefully investigated these cases, but we could not identify the exact cause. Nevertheless, we could address the failure with a high-level intervention that forcibly sets up only one Ethernet interface. More specifically, our intervention forcibly sets up an Ethernet interface, `eth0`, and avoids setting the other interfaces. Thus, we set a bridge network interface and link it to the host if necessary. With this intervention, a large portion of ARM firmware images could be emulated.

Insufficient VLAN setup. VLAN is a typical feature of routers, as it provides an isolated network environment, logically grouping sub networks. A VLAN interface has different characteristics compared to other network interfaces, such as Ethernet or WLAN, and thus it must be set with additional options. To support VLAN, the type of TAP interface should be set to VLAN, and an appropriate VLAN id should be assigned to it.

Another failure occurs in firmware images with VLAN interfaces. When emulating these images, even though Ethernet interfaces were properly configured with independent IP addresses, the guest network was unreachable. Firmadyne attempts to address this by running a command when setting the host TAP interface; however, their configuration is insufficient to handle it. In particular, the VLAN should be set to group the host and guest networks with the same VLAN id. However, Firmadyne dismissed setting the host network. FirmAE arbitrates this by properly configuring the VLAN.

Filtering rules in iptables. Numerous routers set a firewall to prevent unauthorized remote access by design. Otherwise, an attacker could access administration interfaces. Some firmware images in our dataset also implement this policy by using `iptables`. Consequently, the guest kernel drops all packets from the host. We found most of these cases in TP-Link, where the guest is not reachable, even though host and guest networks are configured properly.

This does not represent an emulation failure, since setting iptables mimics the original behavior of real devices. However, such filtering prevents the analysis of their potential vulnerabilities and threats. Evidently, identified vulnerabilities during the analysis might not be remotely exploitable. Nevertheless, numerous device owners or administrators mistakenly change these rules, making the device publicly accessible [14, 15, 51].

FirmAE arbitrates this by checking filtering rules in the guest system and removing them if they exist. This could be done simply by flushing all policies in the `iptables` and setting the default policy to accept all incoming packets. Then, the guest network becomes reachable from the host, and dynamic analysis can be conducted.

4.3 NVRAM arbitrations

Emulating peripherals similar to the real environment is one of the most challenging parts in firmware emulation (§2.3). An NVRAM, which is essentially a flash memory, is one of the peripherals widely used in embedded devices to store configuration data. Internal programs in embedded devices often store/fetch necessary information in/from it. These programs often crash unless NVRAM is supported.

Firmadyne implements a custom NVRAM library to emulate NVRAM-related functions. This custom library is loaded in advance to include other libraries by setting the environmental variable called `LD_PRELOAD`. This intercepts NVRAM-related functions such as `nvram_get()` and `nvram_set()`, and emulates an NVRAM without physical access. Specifically, when `nvram_set()` is called, a key-value pair is stored in a file, and it is later fetched when `nvram_get()` is called. For these cases, where `nvram_get()` is called before the call of `nvram_set()`, Firmadyne initializes key-value pairs using default files in the given firmware, which typically exist for the factory reset functionality of a device. Firmadyne has a list of few hard-coded paths of default files to extract key-value pairs. However, many firmware images in our dataset are still not emulated.

Supporting custom NVRAM default files. We found numerous cases, where the paths of default file differ depending on each device, and even their key-value pairs have different patterns. For example, in some D-Link images, default files are located at `/etc/nvram.default` or `/mnt/nvram_rt.default`. Furthermore, default files in some NETGEAR images are found at `/usr/etc/default`. The key-value pairs in these files are separated with a diverse delimiter, such as a carriage return or `NULL` byte. Some default files even have vendor specific formats, such as `OBJ` or `ELM`.

To develop a scalable approach, FirmAE prepares arbitration during the pre-emulation. Specifically, FirmAE records all the key-value pairs accessed with the `nvram_get()` and `nvram_set()` functions during the pre-emulation. Then, it scans the filesystem of the target firmware and searches files that contain multiple instances of the recorded key names, whose values are unknown. FirmAE extracts the key-value pairs from the files (if they exist) and utilizes them in the final emulation.

No NVRAM default file. Unfortunately, not all firmware images have default NVRAM files. Even if a default file exists, it may not contain the requested key-value pairs. One simple approach to address this issue is to return the `NULL` value for uninitialized keys, as Firmadyne does. However, we observed many cases that crash with a segmentation fault after `nvram_get()` returns `NULL`. By reverse engineering the crashed programs, we found that, surprisingly, many programs do not verify the return value of `nvram_get()`. They just pass the return value into string-related functions, such as `strcpy()` or `strtok()` and crash with a `NULL` pointer dereference.

FirmAE handles this by arbitrating the behavior of the `nvram_get()` function. Instead of returning the `NULL` value when accessing uninitialized keys, FirmAE returns a pointer to an empty string. This simple change significantly decreases crashes, particularly in NETGEAR images. Because we cannot obtain real key-value pairs without physical devices, this would be one of the most optimal approaches to avoid crashes caused by deficient error handling in many internal programs.

4.4 Kernel arbitrations

Many programs in an embedded device co-operate with peripherals through device drivers in the kernel. Typically, they communicate

with peripherals using `ioctl` commands. Unfortunately, emulating this procedure is not a simple task, as each device driver has distinctive characteristics depending on its developers and a corresponding device. Although Firmadyne implemented a few dummy kernel modules, which support `/dev/nvram` and `/dev/acos_nat_cli`, it could not cover diverse characteristics of firmware images in practical scenarios. Many firmware images in our dataset also crash due to this problem.

Insufficient support of kernel module. Since Firmadyne implemented dummy modules with hard-coded device names and `ioctl` commands, some programs fail when accessing kernel modules with a different configuration. For example, numerous NETGEAR images utilize a module called `acos_nat`, which is used to communicate with a peripheral device mounted on `/dev/acos_nat_cli`. In those images, a Firmadyne module returns incorrect values and causes an infinite loop on the web services of `httpd`. Furthermore, we found that `ioctl` commands vary depending on firmware architectures, thus this should be considered as well.

FirmAE's high-level approach takes advantage of emulating a specific kernel module. The key intuition here is that numerous kernel modules are accessed through shared libraries, which have functions that send corresponding `ioctl` commands. Thus, FirmAE intercepts library function calls similarly to handling NVRAM issues (§4.3). When a program calls library functions, FirmAE returns a pre-defined value. Hence, each `ioctl` command does not need to be emulated depending on the device architecture. In this example, we only focused on `acos_nat`, whereas other peripheral accesses via shared libraries can be handled in the same manner.

Improper kernel version. We found some firmware images facing issues with the kernel version. Firmadyne customized Linux kernel v2.6.32 in the firmware emulation. However, recent embedded devices use a newer version of the kernel. Upgrading the kernel version seems like a trivial solution to this problem. Indeed, we experimentally tested Linux kernel v4.1.17 and successfully emulated more firmware images. However, some firmware images, particularly older ones, were not emulated with the new version of the kernel. These images failed with a crash in the `libc` library.

We investigated these cases and determined that the address space layout randomization of Linux kernel v4.1.17 is not compatible with the old versions of `libc`. To resolve this, we used the compatibility option when compiling the new kernel. Specifically, we set the `CONFIG_COMPAT_BRK` option, which excludes randomizing `brk` area in heap memory. With this new kernel, FirmAE was able to handle the above cases. Other compatibility issues may exist that were not detected in our experiment. To address these, multiple kernel versions with various compiling options should be tested further, which is one of the aims of our future studies.

4.5 Other arbitrations

Some failure cases are addressed by other minor interventions.

Unexecuted web servers. For the dynamic analysis of the web service, we need to achieve both network reachability and web service availability. In some images, a web server does not run even after the network is configured successfully. We could not find the exact root cause of this phenomenon. However, an intervention that forcibly executes a web server could address the issue. Specifically, it searches a widely used web server such as `httpd`, `lighttpd`, `boa`, or `goahead` in the filesystem of the target firmware, along with their corresponding configuration files, and executes it.

Timeout issues. Emulating firmware images that do not respond for a long time should be forcibly stopped. Thus, setting a suitable timeout is necessary. Firmadyne applied use a 60 s timeout; however, firmware images, particularly from NETGEAR, take a long time to complete their booting procedure, whereby their emulation is eventually blocked. We investigated such cases and empirically found a suitable timeout of 240 s. Although this change was simple, more than 60 firmware images were successfully emulated.

Lack of tools for emulation. Embedded device developers often omit unnecessary functionalities to save storage. Thus, a firmware image may not have the appropriate tools to emulate itself. As the emulated environment does not have any storage limitation, we can add several required tools. For successful emulation, several Linux commands such as `mount` or `ln` should be prepared in the filesystem. We resolve this by adding the latest version of `busybox` into the filesystem of the target firmware. This simple addition enables essential commands, and leads to successful emulation.

5 EVALUATION

From the investigation on `AnalysisSet`, we found several arbitration points (§4). In this section, we evaluate each arbitration with our prototype FirmAE (§3.3) on our datasets. For this, we implement a total of 3671 LoC in Python and shell scripts. We also introduce the vulnerabilities identified during the dynamic analysis with FirmAE.

5.1 Firmware emulation result

We compare the emulation rates of FirmAE and Firmadyne on each dataset (§3.4). The total emulation time of all datasets was less than four hours (14289 s), as FirmAE supports full automation and parallelization (§3.3).

Overall result. As our goal is to emulate the web services for dynamic analysis (§3.1), we verify the network reachability and web service availability of each emulated firmware. Henceforth, we refer to the web service availability as the emulation rate. The final results are listed in Table 1. Overall, the emulation rate significantly increased from 16.28% to 79.36% (by 487%). Because our investigation is based on `AnalysisSet`, it shows the highest rate of 91.83%. The rates of `LatestSet` and `CamSet` also show a large improvement compared to those obtained by Firmadyne, and we could identify vulnerabilities in them (§5.3). In `AnalysisSet`, the emulation rate of NETGEAR images increased the most, from 10.95% to 93.80% (by 857%), owing to the intervention that addresses ARM network issues, as majority of the NETGEAR images are ARM-based. The emulation rates of TRENDnet, ASUS, Belkin, and Zyxel in `LatestSet` are under 60%; these lower rates are attributed to the larger number of kernel modules in these images and the use of custom hardware interfaces. We describe this in detail in §5.2.

The emulation rates of `CamSet` indicate that addressing failure issues of wireless routers can also help emulate IP cameras. In particular, none of the D-Link images were emulated with Firmadyne, whereas FirmAE could emulate more than 65% of the images. Nevertheless, FirmAE fails to emulate all TP-Link images. We investigated these failed cases and found that they do not contain web servers.

Table 1: Emulation rate of network and web services

Dataset	Vendor	Images	Firmadyne Net	Firmadyne Web	FirmAE Net	FirmAE Web
AnalysisSet	D-Link	179	55	54 (30.17%)	177	167 (93.30%)
	TP-Link	73	26	5 (6.85%)	73	59 (80.82%)
	NETGEAR	274	86	30 (10.95%)	259	257 (93.80%)
Sub Total		526	167	89 (16.92%)	509	483 (91.83%)
LatestSet	D-Link	58	18	17 (29.31%)	54	48 (82.76%)
	TP-Link	69	33	10 (14.49%)	69	54 (78.26%)
	NETGEAR	101	30	7 (6.93%)	92	79 (78.22%)
	TRENDnet	106	35	23 (21.70%)	91	63 (59.43%)
	ASUS	107	27	25 (23.36%)	63	62 (57.94%)
	Belkin	37	2	2 (5.41%)	30	22 (59.46%)
	Linksys	55	13	8 (14.55%)	48	44 (80.00%)
	Zyxel	20	3	0 (0.00%)	18	10 (50.00%)
Sub Total		553	161	92 (16.64%)	465	382 (69.08%)
CamSet	D-Link	26	0	0 (0.00%)	19	17 (65.38%)
	TP-Link	6	0	0 (0.00%)	6	0 (0.00%)
	TRENDnet	13	2	2 (15.38%)	10	10 (76.92%)
Sub Total		45	2	2 (4.44%)	35	27 (60.00%)
Total		**1124**	**330**	**183 (16.28%)**	**1009**	**892 (79.36%)**

The result of CamSet demonstrates that many IP cameras share similar characteristics with wireless routers, such that arbitrations of wireless routers can also be applied to IP cameras.

Impact of each arbitration. We also investigate the effectiveness of each arbitration by omitting a specific arbitration from the final version of FirmAE, to which all arbitrations are applied. This is because numerous arbitration points should co-operate to address the failure, and deducting a specific arbitration directly affects the emulation rate. Figure 2 illustrates these results, and a detailed version is provided in Table 5 in Appendix.

NVRAM arbitration appears to be the most effective, decreasing the emulation rate by 35% on average across all datasets. This coincides with the Firmadyne approach to focus on emulating the NVRAM. Removing the boot and network arbitrations also significantly lowers the emulation rate by ~30%. Only 4.88% of firmware images across all datasets were affected without the kernel arbitration. The other arbitrations affected 22.35% of the firmware images. These results demonstrate that the proposed arbitrations are indeed effective and scalable for successful firmware emulation.

5.2 Post-emulation analysis

Following a large-scale emulation, we investigate unhandled failure issues that cannot be easily addressed by simple arbitrations, but require a more complicated virtualization.

Kernel modules. As discussed in previous studies [12, 17, 23, 25, 57], emulating kernel modules is challenging because 1) different kernel versions often produce compatibility problems, and 2) some firmware images may not have a kernel, such that no useful information can be obtained. In a few cases, web servers and other programs access kernel modules under the /proc directory. Because such files do not exist in the emulated environment, those programs often crash. For example, web servers in TP-Link firmware images access a kernel module at /proc/simple_config/system_code for configuration and subsequently crash, as the module does not exist.

Hardware interfaces. Some internal programs of the firmware use their own dedicated interfaces for peripheral communication, hardening the emulating peripheral interfaces. For example, we hooked popular library calls to emulate a NVRAM. However, some

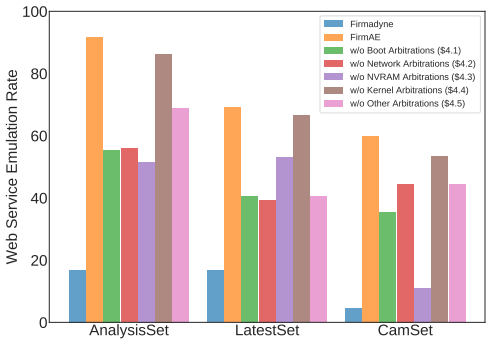

Figure 2: Effectiveness of applied arbitrations

programs of D-Link firmware call /bin/flash to directly access /dev/nvram. Similarly, httpd servers in a few TP-Link firmware images access a flash memory, /dev/ar7100_flash_chrdev, to retrieve information for device configuration. Meanwhile, web servers named webs in Linksys firmware directly manipulate the /dev/mtd interface. They even verify the integrity of the device and verify the signatures and versions of the given firmware.

CGI errors. Even though web servers are accessible, some of them rarely respond with a server error, i.e, *500 Internal Server Error*. There are several causes for this error, such as syntax/code errors in CGI programs, invalid web interface configuration, and PHP errors. However, most error cases are derived by crashes of backend CGI programs. We analyzed the CGI programs with reverse engineering and found that they share the same issues of hardware interfaces. Hence, they attempt to access entries under /proc or /dev to obtain configuration values and stop abnormally if they fail.

The aforementioned cases present the difficulty of emulating peripheral communication without physical devices. Addressing those issues requires a more complicated emulation environment, which is to be addressed in future research.

5.3 Applying Dynamic Analysis – Fuzzing

After having a successful emulation of firmware images, we apply dynamic analysis, fuzzing, to their web services. With this evaluation, we 1) verify that the arbitrated emulation is indeed practical for applying dynamic security analysis of embedded devices, and 2) evaluate the current status of the security of embedded devices in the wild. We target LatestSet and CamSet with the latest firmware.

Dynamic analysis engine. For a large-scale analysis, we focus on the scalability. So our dynamic analysis tool need to be applicable to diverse emulated firmware images with little user interaction. With these criteria, we first searched existing tools [9, 19, 21, 34, 39, 43, 53, 55, 56, 64] and checked if they are applicable to FirmAE.

However, the existing tools do not satisfy our criteria, as they are 1) not publicly available, 2) not scalable for a large-scale analysis, and 3) incapable of finding new vulnerabilities. For example, Firmadyne [17] utilizes Metasploit [43], which checks known vulnerabilities. Other web scanners, such as Burp Suite [55], Arachni [34], or Commix [53] only check a combination of pre-defined HTTP patterns. Thus, they are insufficient for diverse firmware web services in practical scenarios. Furthermore, they are not designed to find memory corruption vulnerabilities, such as buffer

overflow or use-after-free. Meanwhile, the state-of-the-art fuzzer, Firm-AFL [64] is a promising tool to detect memory corruption vulnerabilities. However, it is not applicable to a large-scale analysis, since it requires individual environment setting for a target program. Because of these limitations, we built our own analysis engine. Developing an analysis engine itself is an orthogonal research area, and here we only propose a conceptual design. Our concept may also be applied to the aforementioned tools.

Our analysis engine consists of two parts: it automatically initializes and logs into web pages if necessary, and identifies vulnerabilities including memory corruption bugs. To find 1-day vulnerabilities, we leveraged RouterSploit [56], which has proof-of-concept (PoC) codes of previously known vulnerabilities. We also added several customized PoC codes to it. To analyze 0-day vulnerabilities, we developed a simple web fuzzer with 880 LoC in Python.

Initializing web services. The primary step in dynamic analysis is to initialize web services unless they do not receive any other requests. A large portion of the web services in our dataset require a network and security configuration (e.g., admin or AP password) in the admin pages. However, this initialization procedure also differs in each firmware. Web servers in most firmware images in D-Link, TP-Link, Belkin, Linksys, and ZyXEL automatically initialize themselves after successful emulation, whereas those of ASUS and TRENDnet in particular must be initialized in person. Fortunately, many of them have a skip button to configure default options. Some web services do not explicitly have a skip button, but have internal JavaScript functions that behave identically. Meanwhile, some require a manual admin password.

To automatically process the initialization, we analyzed the initializing process of web services, and extracted representative patterns including buttons and menus from them. Then, we utilize these patterns to automate the process. Here, we leveraged Selenium [50], which is an open-sourced tool that can provide an interface alike a real browser.

Evaluating vulnerability discovery performance. After successfully running the firmware image and its web services, the engine first checks 1-day vulnerabilities utilizing RouterSploit [56] and our customized PoC codes. Because RouterSploit consists of multiple exploits of known vulnerabilities, in this evaluation, we can 1) check if a target device is patched and 2) find a new vulnerable device that is previously unknown, but has the same vulnerability.

To find 0-day vulnerabilities, our engine first searches the filesystem of the target firmware and generates a list of web page candidates by checking the extension of files such as `.html`, `.aspx`, or `.xml`. Then, it extracts possible parameters from the candidates and generates requests to detect vulnerabilities. For example, for the `.htm` and `.html` candidates, our engine parses the HTML tags, such as `script`, `form`, and `input`, to extract target URLs, methods, and parameter information. This approach is particularly helpful when building requests for fuzzing devices that use the home network administration protocol (HNAP); the HNAP request is based on the XML format and the default value is set up in the javascript code of `.html` page. By utilizing the extracted information, we could construct a valid request template for fuzzing. Because we search for candidates from the filesystem, we could also check web services that are not reachable by crawling.

Table 2: 1-day analysis result on `AnalysisSet`

Vulnerability Category	# of PoC	Firmadyne # of Images (Unique)	FirmAE # of Images (Unique)
Information leak	2	0 (0)	17 (17)
Command injection	9	10 (6)	152 (65)
Password disclosure	2	4 (3)	146 (99)
Authentication bypass	2	0 (0)	5 (5)
Total	15	14 (9)	320 (128)

Table 3: New vulnerabilities found on `LatestSet` and `CamSet`

Type	Vulnerability Category	# of Vulns	# of Devices
1-day	Information leak in PHP	1	19
	Information leak in CGI	1	13
	Command injection in UPnP	2	13
	Command injection in SOAP CGI	2	12
	Command injection in HNAP	1	3
	Command injection with backdoor (32764)	2	3
	Path traversal	2	9
	Sub Total	11	72
0-day	Command injection in HNAP	6	13
	Command injection in CGI	1	3
	Buffer overflow in HNAP	1	1
	Buffer overflow in CGI	4	6
	Sub Total	12	23
	Total	23	95

Among the various types of vulnerability, we focus on command injection and buffer overflow as they are often found in embedded devices. To detect command injection vulnerabilities, our engine sends payloads, which are essentially a combination of candidate characters, such as `';'`, `'"'`, or `'&'`, followed by a shell command executing our executable binary. We place this binary to log useful information, such as time and environment variables, thereby checking if the vulnerability is triggered. We also hook the `execve` system call, to easily detect if our inputs are injected in the command. For buffer overflow detection, FirmAE provides a feedback when a crash occurs. Note that we must wait after sending a request to a target web service because of the time required to process the request; we empirically determined that 10 to 15 s is sufficient. We also utilize the boundary values, such as a large-sized buffer for fuzzing inputs, as they are more likely to trigger vulnerabilities.

Any bugs reported by our analysis engine must be verified. For this, we added debugging programs such as `strace`, `gdb`, and `gdbserver` to the filesystem of target firmware. Note we could utilize the `ptrace` system call for debugging as we upgraded the kernel version (§4.4). We also added `netcat` and `telnetd` to access the guest shell. With these tools, we manually verified the identified bugs.

5.4 Dynamic analysis result

To evaluate the effectiveness of arbitrated emulation, we performed a dynamic analysis on each emulated firmware image, of which web services are already initialized by our engine. In particular, after the web service of the target firmware image is initialized by each of FirmAE and Firmadyne, we ran the previously known PoC exploits. We first tested known vulnerabilities using RouterSploit [56] on the emulated images in `AnalysisSet` with FirmAE and Firmadyne each, and the results are listed in Table 2. Without using any arbitration (*i.e.*, Firmadyne), we could only check vulnerabilities in 14 images, of which 9 are unique devices. By applying all the proposed arbitrations (*i.e.*, FirmAE), we could check

vulnerabilities in 320 images, of which 128 are unique. As FirmAE aims to emulate web services (§3.1), all the identified vulnerabilities are located in web services such as SOAP CGI, UPnP, and HNAP. This result shows that FirmAE's successful emulation is helpful to outperform Firmadyne in dynamically analyzing firmware images.

Additionally, we conducted a dynamic analysis including a fuzzer on the latest images in `LatestSet` and `CamSet`. As a result, we found a total of 23 unique vulnerabilities across 95 unique devices. These include 11 1-day and 12 0-day vulnerabilities as listed in Table 3. For the fuzzer, each fuzzing request took an average of 10–15 s when running 50 images in parallel, and the average time spent for finding each vulnerability was 70 min, with the maximum of 150 min. The fuzzing throughput can vary according to the system spec and the number of parallel emulation instances.

An interesting point is that some vendors share the same vulnerabilities. For example, some devices in D-Link and TRENDnet have the same vulnerabilities of information leak, as well as command injection in UPnP and SOAP CGI programs. On the contrary, some NETGEAR devices share a path traversal vulnerability with Xiongmai's. Another point is that the analysis of a target web service may reveal vulnerabilities of other programs related to it. Specifically, when we sent a long payload to detect buffer overflow, a target CGI program stored the payload in a file. Then, another program that reads the written file crashed due to the overflowed payload. Such vulnerability can be only found in the full-system emulation environment, as the user-mode emulation does not consider the filesystem relationship.

In sum, the results demonstrate that FirmAE is practical for vulnerability analysis. We believe that undiscovered vulnerabilities still exist, which should be investigated in future research.

Responsible disclosure. The detected 0-day vulnerabilities were spread across four vendors. We reported all 12 vulnerabilities to the vendors by December 2019, and it took a maximum of nine months to receive their response.

6 DISCUSSIONS

Emulation discrepancy in arbitrated emulation. FirmAE does not aim to eliminate the discrepancy between the real and emulated environment but aim to run the firmware's web server and correctly serve the web interfaces. This may result in a different behavior than running firmware on hardware. However, for applying dynamic security analysis, what we need to check is whether the 1) vulnerable program runs and 2) accepts a malicious input, and 3) triggering the vulnerability in the program. Although the emulation may be incorrect, these three items can be checkable if 1) we can run the web service of the firmware, 2) send an exploit packet via network, and 3) verify if the exploit has been executed successfully or not. Because our arbitrated emulation can support these, the vulnerability discovered by FirmAE is legitimate and also working in the real device.

Generality of arbitration intervention. Although our heuristics for arbitrated emulation performs better than other works for current firmware images, because we develop the heuristics to handle failure cases empirically, our systematized arbitrated emulation can only handle observed cases and may not applicable to new devices and new configurations. In this regard, we believe that an empirical investigation to find such interventions seems indispensable to handle the convoluted nature of IoT devices and their configurations. To encourage future research, we release our code, in the belief that our empirical findings can serve as a reference.

Applying other dynamic analysis techniques. In this study, we developed a simple analysis engine that automatically initializes, logs into, and analyzes web services for dynamic analysis. However, each step can be further improved by applying other promising techniques. For example, the login procedure may be analyzed and bypassed by using symbolic execution [52]. Moreover, adopting other fuzzing strategies [8, 48], hybrid analysis approaches [54, 61], or similarity techniques [18, 20], may discover even more vulnerabilities. We leave such promising improvements on the dynamic analysis engine as a future work.

Applying emulation to build an IoT honeypot. Arbitrated emulation can also be useful to build a honeypot for analyzing numerous attacks targeting IoT devices. In fact, there have been several honeypot studies utilizing emulation [35, 47, 49, 57]. Particularly, Vetterl *et al.* [57] proposed a honeypot named Honware based on firmware emulation similarly to FirmAE's approach. As a honeypot should interact with an attacker outside the network, the authors focused on increasing the network reachability rate by investigating emulation failure cases. Accordingly, FirmAE's network intervention that configures a default network setting is fairly similar to Honware's approach. However, FirmAE includes additional interventions to run web services for actively analyzing vulnerabilities in them, and such interventions even more increased the emulation rate (Table 5). Thus, we believe that arbitrated emulation can be useful to build an IoT honeypot as well.[2]

7 CONCLUSION

Analysis of embedded device security has received considerable attention. In this study, we investigated a large-scale firmware dataset and discovered that firmware emulation can substantially benefit from simple interventions. We proposed arbitrated emulation and interventions that can address high-level failure problems. With a prototype, FirmAE, we demonstrated that the proposed approach can boost the emulation rate of the state-of-the-art framework by 487%. We also performed dynamic analysis on the emulated firmware and found 23 unique vulnerabilities, including 12 0-days.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback, and Minkyo Seo for developing the containerization. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00831, a study on physical layer security for heterogeneous wireless network, and No.2019-0-01343, regional strategic industry convergence security core talent training business)

REFERENCES

[1] 2014. *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

[2] We could not find public source code for Honware for evaluation.

[2] 2016. *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[3] 2019. *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[4] 2020. *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[5] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, BC, Canada.

[6] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*.

[7] Roland Bodenheim, Jonathan Butts, Stephen Dunlap, and Barry Mullins. 2014. Evaluation of the ability of the Shodan search engine to identify Internet-facing industrial control devices. *International Journal of Critical Infrastructure Protection* 7, 2 (2014), 114–123.

[8] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA, 725–741.

[9] Wang Chunlei, Liu Li, and Liu Qiang. 2014. Automatic fuzz testing of web service vulnerability. In *Proceedings of the International Conference on Information and Communications Technologies (ICT 2014)*. IET, Nanjing, China.

[10] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation, See [4].

[11] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares, See [1].

[12] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. 2016. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Xi'an, China.

[13] A Cui. 2012. Embedded Device Firmware Vulnerability Hunting Using FRAK. In *Black Hat USA Briefings (Black Hat USA)*. Las Vegas, NV.

[14] Ang Cui, Michael Costello, and Salvatore J Stolfo. 2013. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Proceedings of the 2013 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[15] Ang Cui and Salvatore J Stolfo. 2010. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[16] CVE 2014. CVE-2014-3936. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3936.

[17] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. 2016. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware, See [2].

[18] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 392–404.

[19] R Dawes. 2011. OWASP WebScarab Project.

[20] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[21] Michael Eddington. 2011. Peach fuzzing platform. *Peach Fuzzer* 34 (2011).

[22] Florian Fainelli. 2008. The OpenWrt embedded development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting*.

[23] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling, See [4].

[24] NCC Group et al. 2017. A linux system call fuzzer using TriforceAFL. https://github.com/nccgroup/TriforceAFL.

[25] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. 2019. Toward the Analysis of Embedded Firmware through Automated Re-hosting. In *Proceedings of the 22th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Beijing, China.

[26] Craig Heffner. 2010. Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk.

[27] Craig Heffner, Jeremy Collake, et al. 2011. Firmware Mod Kit. https://github.com/rampageX/firmware-mod-kit.

[28] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. 2016. Embedded security testing with peripheral device caching and runtime program state approximation. In *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*.

[29] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. Kyoto, Japan.

[30] Swati Khandelwal. 2016. Multiple Backdoors found in D-Link DWR-932 B LTE Router. http://thehackernews.com/2016/09/hacking-d-link-wireless-router.html?m=1.

[31] Swati Khandelwal. 2017. Satori IoT Botnet Exploits Zero-Day to Zombify Huawei Routers. https://thehackernews.com/2017/12/satori-mirai-iot-botnet.html.

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

[33] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. 2019. All things considered: an analysis of IoT devices on home networks, See [3].

[34] Tasos Laskos. 2010. Arachni. http://www.arachni-scanner.com.

[35] Samuel Litchfield, David Formby, Jonathan Rogers, Sakis Meliopoulos, and Raheem Beyah. 2016. Rethinking the honeypot for cyber-physical systems. *IEEE Internet Computing* 20, 5 (2016), 9–17.

[36] Knud Lasse Lueth. 2018. State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating.

[37] David Maciejak. 2018. Yet Another Crypto Mining Botnet? https://www.fortinet.com/blog/threat-research/yet-another-crypto-mining-botnet.html.

[38] Denis Makrushin. 2018. Backdoors in D-Link's backyard. https://securelist.com/backdoors-in-d-links-backyard/85530.

[39] Xavi Mendez. 2014. wfuzz. https://github.com/xmendez/wfuzz.

[40] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[41] Ryan Mitchell. 2018. *Web Scraping with Python: Collecting More Data from the Modern Web*. "O'Reilly Media, Inc.".

[42] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[43] HD Moore et al. 2009. The Metasploit project. https://www.metasploit.com.

[44] Marius Muench, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (BAR)*.

[45] 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 *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[46] Jeong Wook Oh. 2014. Reverse engineering flash memory for fun and benefit. In *Black Hat USA Briefings (Black Hat USA)*. Las Vegas, NV.

[47] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. 2015. IoTPOT: analysing the rise of IoT compromises. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*. Washington, DC.

[48] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing, See [1].

[49] Lukas Rist, Johnny Vestergaard, Daniel Haslinger, Andrea Pasquale, and John Smith. 2013. Conpot ics/scada honeypot. http://conpot.org.

[50] Selenium 2004. Selenium. https://www.seleniumhq.org.

[51] Shodan. 2016. D-Link Internet Report. https://dlink-report.shodan.io/.

[52] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[53] Anastasios Stasinopoulos, Christoforos Ntantogian, and Christos Xenakis. 2015. Commix: Detecting and exploiting command injection flaws. In *Black Hat USA Briefings (Black Hat USA)*. Las Vegas, NV.

[54] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution., See [2].

[55] Dafydd Stuttard. 2008. Burp Suite. https://portswigger.net/burp.

[56] Threat9. 2016. RouterSploit. https://github.com/threat9/routersploit.

[57] Alexander Vetterl and Richard Clayton. 2019. Honware: A virtual honeypot framework for capturing CPE and IoT zero days. In *2019 APWG Symposium on Electronic Crime Research (eCrime)*. IEEE, 1–13.

[58] Wikipedia contributors. 2018. IP aliasing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=IP_aliasing&oldid=871887325. [Online; accessed 13-August-2019].

[59] Matt Wilson. 2019. Premium Wireless Routers Market Size, Share, Statistics, Trends, Types, Applications, Analysis and Forecast| Global Industry Research and Forecast 2019-2024. https://marketersmedia.com/premium-wireless-routers-market-size-share-statistics-trends-types-applications-analysis-and-forecast-global-industry-research-and-forecast-2019-2024/520294.

[60] Heng Yin Xunchao Hu, Yaowen Zheng. 2018. An Extensible Dynamic Analysis Framework for IoT Devices. In *Black Hat USA Briefings (Black Hat USA)*. Las Vegas, NV.
[61] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, 745–761.
[62] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[63] Michal Zalewski. 2017. American fuzzy lop (AFL). http://lcamtuf.coredump.cx/afl. (2017).
[64] 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, See [3], 1099–1114.

Table 4: Full statistics of firmware dataset

Dataset	Vendor	# of Images	# of Architecture				# of Web Services								
			arm32el	mips32el	mips32eb	etc	httpd	uhttpd	mini_httpd	lighttpd	alphapd	goahead	boa	jjhttpd	etc
AnalysisSet	D-Link	179	22 (12.29%)	82 (45.81%)	75 (41.90%)	0 (0.00%)	102 (56.98%)	0 (0.00%)	0 (0.00%)	9 (5.03%)	39 (21.79%)	10 (5.59%)	2 (1.12%)	14 (7.82%)	5 (2.79%)
	TP-Link	73	10 (13.70%)	15 (20.55%)	48 (65.75%)	0 (0.00%)	64 (87.67%)	9 (12.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	NETGEAR	274	105 (38.32%)	56 (20.44%)	113 (41.24%)	0 (0.00%)	125 (45.62%)	77 (28.10%)	69 (25.18%)	5 (1.82%)	0 (0.00%)	0 (0.00%)	3 (1.09%)	0 (0.00%)	0 (0.00%)
Sub Total		526	137 (26.05%)	153 (29.09%)	236 (44.87%)	0 (0.00%)	291 (55.32%)	86 (16.35%)	69 (13.12%)	14 (2.66%)	39 (7.41%)	10 (1.90%)	5 (0.95%)	14 (2.66%)	5 (0.95%)
LatestSet	D-Link	58	9 (15.52%)	17 (29.31%)	32 (55.17%)	6 (10.34%)	39 (67.24%)	0 (0.00%)	8 (13.79%)	12 (20.69%)	0 (0.00%)	1 (1.72%)	4 (6.90%)	3 (5.17%)	0 (0.00%)
	TP-Link	69	13 (18.84%)	22 (31.88%)	34 (49.28%)	0 (0.00%)	53 (76.81%)	16 (23.19%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	NETGEAR	101	32 (31.68%)	44 (43.56%)	25 (24.75%)	1 (0.99%)	46 (45.54%)	36 (35.64%)	19 (18.81%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	2 (1.98%)	0 (0.00%)	0 (0.00%)
	TRENDnet	106	18 (16.98%)	29 (27.36%)	59 (55.66%)	0 (0.00%)	28 (26.42%)	9 (8.49%)	13 (12.26%)	6 (5.66%)	0 (0.00%)	11 (10.38%)	11 (10.38%)	3 (2.83%)	21 (19.81%)
	ASUS	107	28 (26.17%)	72 (67.29%)	2 (1.87%)	0 (0.00%)	106 (99.07%)	0 (0.00%)	0 (0.00%)	51 (47.66%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	1 (0.93%)
	Belkin	37	2 (5.41%)	20 (54.05%)	15 (40.54%)	0 (0.00%)	25 (67.57%)	0 (0.00%)	11 (29.73%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	4 (10.81%)
	Linksys	55	15 (27.27%)	30 (54.55%)	10 (18.18%)	1 (1.82%)	23 (41.82%)	1 (1.82%)	6 (10.91%)	26 (47.27%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
	Zyxel	20	5 (25.00%)	10 (50.00%)	5 (25.00%)	0 (0.00%)	2 (10.00%)	0 (0.00%)	2 (10.00%)	7 (35.00%)	0 (0.00%)	2 (10.00%)	5 (25.00%)	0 (0.00%)	3 (15.00%)
Sub Total		553	122 (22.06%)	244 (44.12%)	182 (32.91%)	8 (1.45%)	322 (58.23%)	62 (11.21%)	59 (10.67%)	102 (18.44%)	0 (0.00%)	14 (2.53%)	22 (3.98%)	6 (1.08%)	29 (5.24%)
CamSet	D-Link	26	8 (30.77%)	15 (57.69%)	3 (11.54%)	0 (0.00%)	6 (23.08%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	13 (50.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (23.08%)
	TP-Link	6	6 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (100.00%)
	TRENDnet	13	1 (7.69%)	10 (76.92%)	2 (15.38%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (46.15%)	0 (0.00%)	2 (15.38%)	0 (0.00%)	2 (15.38%)
Sub Total		45	15 (33.33%)	25 (55.56%)	5 (11.11%)	0 (0.00%)	6 (13.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	19 (42.22%)	0 (0.00%)	2 (4.44%)	0 (0.00%)	14 (31.11%)
Total		**1124**	**274 (24.38%)**	**422 (37.54%)**	**423 (37.63%)**	**8 (0.71%)**	**619 (55.07%)**	**148 (13.17%)**	**128 (11.39%)**	**116 (10.32%)**	**58 (5.16%)**	**24 (2.14%)**	**29 (2.58%)**	**20 (1.78%)**	**48 (4.27%)**

Table 5: Full result of FirmAE removing each arbitration

Dataset	Vendor	# of Images	FirmAE		w/o Boot Arbitrations		w/o Network Arbitrations		w/o NVRAM Arbitrations		w/o Kernel Arbitrations		w/o Other Arbitrations	
			Network	Web Service	Network	Web Service	Network	Web Service	Network	Web Service	Network	Web Service	Network	Web Service
AnalysisSet	D-Link	179	177 (98.88%)	167 (93.30%)	162 (90.50%)	145 (81.01%)	100 (55.87%)	90 (50.28%)	176 (98.32%)	129 (72.07%)	154 (86.03%)	144 (80.45%)	173 (96.65%)	146 (81.56%)
	TP-Link	73	73 (100.00%)	59 (80.82%)	53 (72.60%)	36 (49.32%)	27 (36.99%)	13 (17.81%)	73 (100.00%)	56 (76.71%)	73 (100.00%)	60 (82.19%)	55 (75.34%)	31 (42.47%)
	NETGEAR	274	259 (94.53%)	257 (93.80%)	110 (40.15%)	110 (40.15%)	191 (69.71%)	191 (69.71%)	239 (87.23%)	86 (31.39%)	259 (94.53%)	250 (91.24%)	252 (91.97%)	185 (67.52%)
Sub Total		526	509 (96.77%)	483 (91.83%)	325 (61.79%)	291 (55.32%)	318 (60.46%)	294 (55.89%)	488 (92.78%)	271 (51.52%)	486 (92.40%)	454 (86.31%)	480 (91.25%)	362 (68.82%)
LatestSet	D-Link	58	54 (93.10%)	48 (82.76%)	46 (79.31%)	41 (70.69%)	19 (32.76%)	18 (31.03%)	54 (93.10%)	48 (82.76%)	54 (93.10%)	40 (68.97%)	51 (87.93%)	45 (77.59%)
	TP-Link	69	69 (100.00%)	54 (78.26%)	54 (78.26%)	32 (46.38%)	39 (56.52%)	23 (33.33%)	69 (100.00%)	53 (76.81%)	69 (100.00%)	57 (82.61%)	57 (82.61%)	23 (33.33%)
	NETGEAR	101	92 (91.09%)	79 (78.22%)	49 (48.51%)	41 (40.59%)	68 (67.33%)	60 (59.41%)	92 (91.09%)	25 (24.75%)	92 (91.09%)	82 (81.19%)	87 (86.14%)	54 (53.47%)
	TRENDnet	106	91 (85.85%)	63 (59.43%)	55 (51.89%)	41 (38.68%)	49 (46.23%)	37 (34.91%)	91 (85.85%)	56 (52.83%)	87 (82.08%)	52 (49.06%)	84 (79.25%)	44 (41.51%)
	ASUS	107	63 (58.88%)	62 (57.94%)	31 (28.97%)	31 (28.97%)	34 (31.78%)	32 (29.91%)	63 (58.88%)	45 (42.06%)	62 (57.94%)	61 (57.01%)	58 (54.21%)	25 (23.36%)
	Belkin	37	30 (81.08%)	22 (59.46%)	3 (8.11%)	3 (8.11%)	14 (37.84%)	14 (37.84%)	30 (81.08%)	19 (51.35%)	30 (81.08%)	22 (59.46%)	29 (78.38%)	5 (13.51%)
	Linksys	55	48 (87.27%)	44 (80.00%)	34 (61.82%)	34 (61.82%)	31 (56.36%)	31 (56.36%)	47 (85.45%)	42 (76.36%)	48 (87.27%)	44 (80.00%)	44 (80.00%)	27 (49.09%)
	Zyxel	20	18 (90.00%)	10 (50.00%)	7 (35.00%)	2 (10.00%)	8 (40.00%)	2 (10.00%)	18 (90.00%)	6 (30.00%)	18 (90.00%)	10 (50.00%)	18 (90.00%)	1 (5.00%)
Sub Total		553	465 (84.09%)	382 (69.08%)	279 (50.45%)	225 (40.69%)	262 (47.38%)	217 (39.24%)	464 (83.91%)	294 (53.16%)	460 (83.18%)	368 (66.55%)	428 (77.40%)	224 (40.51%)
CamSet	D-Link	26	19 (73.08%)	17 (65.38%)	13 (50.00%)	12 (46.15%)	13 (50.00%)	11 (42.31%)	18 (69.23%)	3 (11.54%)	18 (69.23%)	16 (61.54%)	18 (69.23%)	14 (53.85%)
	TP-Link	6	6 (100.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	6 (100.00%)	0 (0.00%)	6 (100.00%)	0 (0.00%)	6 (100.00%)	0 (0.00%)
	TRENDnet	13	10 (76.92%)	10 (76.92%)	10 (76.92%)	4 (30.77%)	10 (76.92%)	9 (69.23%)	10 (76.92%)	2 (15.38%)	10 (76.92%)	8 (61.54%)	10 (76.92%)	6 (46.15%)
Sub Total		45	35 (77.78%)	27 (60.00%)	23 (51.11%)	16 (35.56%)	23 (51.11%)	20 (44.44%)	34 (75.56%)	5 (11.11%)	34 (75.56%)	24 (53.33%)	34 (75.56%)	20 (44.44%)
Total		**1124**	**1009 (89.77%)**	**892 (79.36%)**	**627 (55.78%)**	**532 (47.33%)**	**603 (53.65%)**	**531 (47.24%)**	**986 (87.72%)**	**570 (50.71%)**	**980 (87.19%)**	**846 (75.27%)**	**942 (83.81%)**	**606 (53.91%)**