Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation

Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, Adam Doupé, Tiffany Bao, Yan Shoshitaishvili, and Ruoyu Wang, *Arizona State University*

https://www.usenix.org/conference/usenixsecurity23/presentation/tay

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation

Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy,
Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, Adam Doupé,
Tiffany Bao, Yan Shoshitaishvili, Ruoyu Wang

Arizona State University
{htay2,zengyhkyle,jvadayat,arvindsraj,dutcher,vsiginen,
wfgibbs,zbasque,fdong12,zesmith1,doupe,tbao,yans,fishw}@asu.edu

Abstract

As IoT devices grow more widespread, scaling current analysis techniques to match becomes an increasingly critical task. Part of this challenge involves not only rehosting the firmware of these embedded devices in an emulated environment, but to do so and discover real vulnerabilities. Current state-of-the-art approaches for rehosting must account for the discrepancies between emulated and physical devices, and thus generally focus on improving *the emulation fidelity*. However, this pursuit of fidelity ignores other potential solutions.

In this paper, we propose a novel rehosting technique, *user-space single-service rehosting*, which emulates a single firmware service in user space. We study the rehosting process involved in hundreds of firmware samples to generalize a set of *roadblocks* that prevent emulation and create *interventions* to resolve them. Our prototype Greenhouse automatically rehosts 2,841 (39.7%) of our collected 7,140 firmware images from nine different vendors. Our approach sidesteps many of the challenges encountered by previous rehosting techniques and enables us to apply common vulnerability discovery techniques to our rehosted images such as user-space coverage-guided fuzzing. Using these techniques, we find 717 N-day vulnerabilities and 26 zero-day vulnerabilities on a subset of our rehosted firmware services.

1 Introduction

The Internet of Things (IoT) outnumbers humans almost 2-to-1: As of 2022, approximately 14.4 billion connected IoT devices [18] exist out in the wild, and current estimates project the total number to reach 34.2 billion by 2025 [21]. Naturally, the security of these devices is not perfect, with 747 vulnerabilities across 86 different vendors disclosed in the first half of 2022 alone [34]. The *actual* number of vulnerabilities, undiscovered, unreported, and lurking in IoT firmware, is almost certainly much higher.

To discover latent vulnerabilities in IoT devices, researchers look to apply program analysis techniques, including dynamic approaches such as web scanning and coverage-guided fuzzing, on IoT device firmware. However, such attempts are often curtailed by the inaccessibility of IoT devices: Purchasing them does not scale and can be time-consuming, overly expensive, or even impossible. Even with physical access to IoT devices, the rigidity of hardware, operating systems, and applications on such devices usually make applying aforementioned dynamic analysis techniques very difficult.

A common solution to address this problem is firmware rehosting, or rehosting for short, which emulates IoT software on powerful, flexible, non-IoT devices, such as personal computers (PCs) and servers. A key challenge in rehosting is the high-fidelity emulation of characteristics and features that are specific to each IoT device. For example, IoT firmware commonly stores data in NVRAM (non-volatile random-access memory), which does not exist on most x86 PCs and must be emulated by the rehosting environment. Peripherals can pose difficulties, too. A software service on a wireless router may send and receive radio signals using antennas that only exist on the router, and a full emulation of behaviors of antennas usually requires significant manual effort.

Seeking high rehosting fidelity, researchers have proposed techniques to either emulate peripherals [4, 14, 19, 22, 28] or proxy the communication to peripherals running on real IoT devices [17, 27, 33]. However, most current peripheral-aware rehosting techniques only allow for the analysis of firmware based on small, embedded software platforms (such as FreeRTOS [2] or Arduino [3]) or no operating system (OS) at all ("bare-metal" blobs). Critically, current approaches cannot scale to the analysis of more complex Linux-based firmware, the OS that is used by 43% of IoT devices [15].

State-of-the-art rehosting solutions that target Linux-based firmware rely on peripheral-oblivious full-system rehosting, typically by repacking the firmware sample into a standard filesystem format, replacing the embedded Linux kernel with a rehost-specific version to support some ad hoc generic device emulation, and booting the firmware sample in a full-system emulator such as QEMU [5, 21]. However, this implicit concession of rehosting fidelity (e.g., by replacing the embedded kernel) leads to rehosting failures. For example,

Firmadyne [5] only achieves IP connectivity on 21% of attempted firmware samples. Even for firmware that *is* ostensibly properly rehosted, lack of fidelity leads to errors in firmware operation: FirmAE [21], a refined version of Firmadyne, measures a successful rehosting rate of 79%, but we show in this paper that almost half of FirmAE-rehosted targets actually do not maintain sufficient functionality to test externally-facing services.

One clear research direction to mitigate rehosting failures is to *increase* fidelity. But is this pursuit of fidelity in emulation a must for rehosting? We performed a random sampling of 100 firmware CVEs reported on NVD [25] in the last two years and found that only 14% of them were hardware related, and many of the remaining ones are intrinsically independent of hard-to-emulate, device-specific characteristics and features. For example, in September 2022, Tenda disclosed 10 Buffer Overflow vulnerabilities (CVE-2022-40067 to CVE-2022-40076) in network-facing functions of the `httpd` binary in its Linux-based AC21 device firmware, and none of these vulnerabilities require interactions with peripherals. For the purpose of vulnerability discovery and vulnerability verification on IoT software, it is often unnecessary to achieve high-fidelity emulation of these characteristics and features if the vulnerable service can run without them. In fact, the pursuit of fidelity can blind researchers to other potential techniques that might be able to achieve successful purpose-specific rehosting.

In this paper, we propose a novel rehosting technique: Automated single-service rehosting. Unlike other rehosting solutions, single-service rehosting does not mandate high-fidelity emulation of OS or hardware. Instead, we design a series of techniques that automatically find execution barriers during the rehosting of a firmware service, use a toolkit of interventions (e.g., patching the service binary to eliminate certain environment checks) to surmount these barriers, and validate the patched service to check if our patches break intended features. By not emulating OS- or hardware-specific characteristics and features, our solution not only avoids pitfalls encountered by full-system techniques (such as incompatibilities between the inserted rehosted kernel and the embedded system itself), but as a bonus also enables user-space emulation, which significantly reduces the execution overhead that full-system emulation techniques exhibit. Moreover, user-space emulation enables common vulnerability discovery and verification techniques, such as coverage-guided fuzzing.

We develop a platform, Greenhouse, to perform automated rehosting of *single-services* via user-space emulation. Our approach *fully rehosted* 2,841 web servers out of 7,140 crawled firmware images and successfully confirmed 717 N-day exploits using the RouterSploit framework [36]. We also integrate our rehosted images with AFL++ [39] for fuzzing in user space, achieving a throughput that is 200% higher than the state-of-the-art firmware-fuzzing solution, EQUAFL [41]. We then extend this integration to 2,612 targets in our dataset and found 11,395 unique crashes. Of these targets, we further examined 14 rehosted targets with 79 crashes and confirmed 26 to be zero-day vulnerabilities.

Finally, in the course of developing service validation checks for Greenhouse, we found that prior work incorrectly reported non-functional rehosted firmware services as successful rehosting targets. For example, FirmAE performs an HTTP request against rehosted web servers to determine rehosting success, but does not check the content of the webpage or the status code for errors (i.e., only checks if the service responds with *something*). These non-functional services are of limited use in discovering or assessing vulnerabilities. Thus, we also propose new criteria for differentiating among rehosting failures, partially rehosted services, and fully rehosted services. Using this technique, our comparative evaluation shows that Greenhouse is slightly more successful than state-of-the-art work at firmware rehosting, but *rehosts mostly different firmware*, resulting in 3,981 unique rehosted samples when combined with full-system rehosting approaches.

Contributions. In summary, our contributions are:

- We propose a novel rehosting technique, *user-space single-service rehosting*, for rehosting firmware services for the purpose of finding and assessing vulnerabilities on IoT software. We implement this technique in a prototype called Greenhouse.
- We thoroughly study the rehosting process and provide a detailed breakdown of causes (termed *roadblocks*) of emulation failures in user-mode rehosting alongside both generic and specific *interventions*.
- We conduct a large-scale evaluation on 7,140 unique firmware samples from nine different vendors and *fully rehost* 2,841 web servers. We also demonstrate unique advantages of Greenhouse for vulnerability discovery and assessment by comparing against the state-of-the-art firmware fuzzing solution, EQUAFL.

In the spirit of open science, we publicly release the source of Greenhouse and research artifacts at `https://github.com/sefcom/greenhouse`.

2 Background and Motivation

Rehosting is the process of recreating the behaviors of one or several firmware services inside an emulated environment. Given the dire situation of IoT-world security, most firmware rehosting techniques aim to enable security analysis techniques, such as automated vulnerability discovery [14, 22, 27, 32, 40] and vulnerability risk assessment (i.e., assessing the real-world impact on firmware of an exploit) [5, 21, 42] to be performed on firmware targets.

In this section, we first discuss the different rehosting goals of existing works (Section 2.1). Next we define rehosting fidelity (Section 2.2), map the goals and focuses of existing works to differing levels of fidelity, and identify a research gap (Section 2.3). Finally, we motivate our solution that fills

the gap (Section 2.4).

2.1 Rehosting Goals

The rigidity of firmware and its original hardware severely limits the types of analysis that researchers can conduct. Thus, the need to apply automated and scalable security analyses to a firmware service motivates firmware rehosting research [13]. As these firmware services are usually tightly coupled with their software and hardware environment, emulating even a single service may require recreating all underlying hardware and software components. Creating a perfect emulation for every firmware service is complicated by the varied nature of IoT firmware.

Researchers categorize IoT firmware into three types [13, 24]: **Type-I** firmware, which runs general-purpose OSs (e.g., Linux) adapted to embedded environments. **Type-II** firmware, which has custom OSs designed for embedded environments, but still shares a distinction between the application layer and the kernel. **Type-III** firmware, also known as "monolithic firmware," where the code is a single blob running on the device, and interacts with its hardware using specialized interfaces. Because Type-II and Type-III firmware are tightly coupled to its hardware, generalized rehosting tools for Type-II and Type-III are less common compared to Type-I rehosting tools.

Rehosting techniques must provide a virtual environment that is capable of executing a firmware service and minimize discrepancies between a rehosted environment and the original environment (i.e., a real device) based on their analyses. For example, works that look to analyze the security of peripheral communication protocols (e.g., USB) must emulate or integrate these components [19, 27, 33]. Solutions that minimize environment discrepancies may simulate intermediary hardware layers [7, 14].

2.2 Rehosting Fidelity

Divergences between the emulated and the original environments can cause a rehosted service to behave differently or even fail to run. Abstractly, we define the *fidelity* of an emulated firmware component as the degree to which it resembles the same component on a real device. We further coin the fidelity of static components (e.g., files) as *Extraction Fidelity* and the fidelity of dynamic components (e.g., runtime behaviors of a service) as *Execution Fidelity*. In general, the more components resemble their counterparts on the original device, the higher the fidelity of an emulation.

Extraction Fidelity. Static components in a firmware image include files and data that is stored on the image or in hardware (e.g., NVRAM). Researchers usually extract these components from downloaded firmware images and physical devices. A high degree of Extraction Fidelity means that the majority of components on the original device are obtained or extracted, while low Extraction Fidelity arises when extraction fails or yields incomplete results.

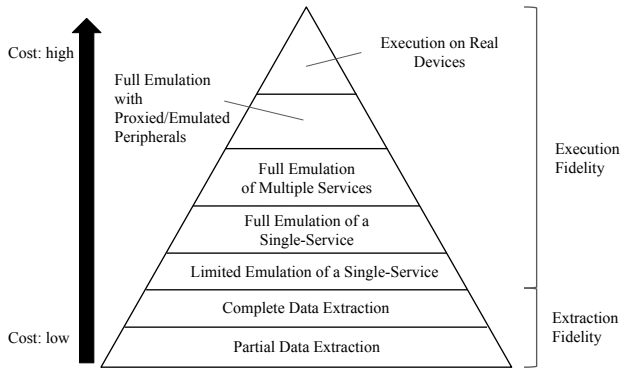

Figure 1: Hierarchy of rehosting requirements. Both rehosting fidelity and costs increase from bottom to top.

Execution Fidelity. Execution Fidelity of a firmware service can be compounded by the presence or absence of *peripherals*—hardware components that communicate with the service to perform specific tasks, such as controlling LEDs and reading sensor data. Examples of low Execution Fidelity in an emulated service include unintended behaviors, broken communications (e.g., failing to log in to an emulated web portal), or prematurely exiting or crashing. Achieving a high level of Execution Fidelity is critical for vulnerability discovery and assessment, as behaviors caused by low Execution Fidelity may lead to false positives (if unintended crashes are treated as vulnerability indicators) or false negative alarms (when vulnerable code does not run in emulation).

Execution Fidelity and Extraction Fidelity. High Extraction Fidelity is critical for high Execution Fidelity, because configuration entries and file content can affect the behavior of firmware services. Figure 1 illustrates the hierarchy of rehosting requirements. The lower levels are more fundamental and easier to achieve, and the upper levels are usually much harder to achieve. Costin [9] considered this hierarchy in terms of emulation level, with a "perfect" emulation corresponding to the top of the rehosting pyramid, followed by original kernel, generic kernel, userland emulation and lastly no emulator (data extraction). Jetset [20] divides it into different approaches, each at increasing levels of fidelity: testing without emulation, partial rehosting, full rehosting and hardware-in-the-loop emulation.

2.3 Existing Approaches

Table 1 categorizes existing rehosting techniques. Most solutions either skew towards achieving a high Execution Fidelity for dynamic analysis or sidestep the issue entirely via pure static analysis [12] or a model-based approach [17, 20].

In cases where the Execution Fidelity of emulation is of concern, researchers tend to use an OS-level emulator like QEMU-system. Even then, configuration, software, and hardware discrepancies between the real device and an emulation environment are often inevitable. Resolving these discrep-

Fidelity	Emulation	Analysis	Firmware Type	Techniques
Full Emulation of Multiple Services + Peripherals	OS-level Emulation (QEMU)	Dynamic Analysis	II/III	μEmu [42], Jetset [20], P2IM [14], HALucinator [7]
Full Emulation of Multiple Services + Peripherals	OS-level Emulation (QEMU)	Dynamic Analysis	I	Charm [33]
Full Emulation of Multiple Services + Peripherals	Model-based Emulation	Dynamic Analysis	II/III	Pretender [17]
Full Emulation of Multiple Services	OS-level Emulation (QEMU)	Dynamic Analysis	I	Costin16 [9], Firmadyne [5], FirmAE [21], FirmFuzz [32]
Full Emulation of Multiple Services	Model-based Emulation	Dynamic Analysis	II/III	Fuzzware [28]
Full Emulation of Multiple Services	Hybrid Emulation (QEMU)	Dynamic Analysis	I	FirmAFL, EQUAFL [41], Frankenstein [27]
Complete Data Extraction	Model-based Emulation	Static Analysis	III	HEAPSTER [16]
Complete Data Extraction	None	Static Analysis	I/II/III	FirmUSB [19], Karonte [12]

Table 1: State-of-the-art rehosting techniques organized by the Execution Fidelity and Extraction Fidelity.

ancies piecemeal requires significant manual effort. Furthermore, OS-level emulation comes with high performance overhead, especially for dynamic analysis techniques like fuzzing.

Alternatively, works like Costin [9] considered the use of user-space emulation for dynamic analysis. Despite possessing lower overhead, rehosting in user-space results in significantly lower fidelity for emulated services, noted to be "quite unstable" by the authors [9].

A workaround that Frankenstein [27], FirmAFL [40] and EQUAFL [41] use is to augment the fidelity of the user-space emulation with a snapshot from a real device or a full-system emulator. This results in a tight coupling between the fuzzer and the rehosted targets, which hampers the generality of rehosting techniques. For example, FirmAFL requires intensive manual effort for harnessing *each* new firmware target (details in Appendix).

2.4 Motivation

Existing techniques imply that firmware rehosting mandates high execution fidelity. However, high execution fidelity is often unnecessary for security analysis of firmware, especially when the target vulnerabilities do not require high fidelity.

We sampled 100 firmware CVEs that are reported on NVD [25] within the past two years and found that only 14% of them were hardware-related. Many of the remaining ones are intrinsically independent of hard-to-emulate, device-specific characteristics and features. For example, Tenda disclosed 10 vulnerabilities in the `httpd` executable of AC21 router, none of which interact with peripherals [35].

Firmware design may require that an emulated service load configuration entries, communicate with a peripheral, or invoke device-specific features, *before* reaching vulnerable program points. Instead of blindly increasing Execution Fidelity, researchers propose to overcome these *roadblocks* by providing low-fidelity substitutes, such as creating limited emulations using stubs [5, 21, 32] or models [14, 17, 42] Our insight is that we can use similar approaches to handle the low-fidelity parts of user-space rehosting. Rather than a crude, high-effort reimplementation of hardware components to increase Execution Fidelity across the entire emulated environment, it is sufficient for security analysis purposes to bypass these roadblocks and only focus on components that are *immediately relevant* to the execution of potentially vulnerable code.

Because most IoT vulnerabilities only involve one firmware service, we focus on user-mode emulation of *individual* firmware services. By recreating only the necessary emulation surrounding a firmware service, we aim to gain the benefits of high Execution Fidelity without expensive full-system emulation. This allows us to significantly improve the execution speed and the portability of rehosted services.

3 Single-Service Rehosting

Greenhouse rehosts Type-I IoT devices (as defined in Section 2.1) that use Linux-based OSs. We select routers because they represent the largest and most commonly studied subset of IoT devices. We limit ourselves to firmware images of the following 32-bit architectures: MIPS, MIPSEL, ARM, and X86, as they represent the majority of publicly available firmware images.

A single service on firmware. Consider the firmware on a device with multiple running processes that constantly exchange information during execution. We can define *services* based on data hierarchy between these processes. A *single service* represents a self-contained set of processes within the image that do not communicate with any other processes that the primary process is not the parent of. For example, a web server may invoke several scripts to dynamically generate HTML content for users. The web server is the primary process, which, together with the scripts, constitutes a single service.

Greenhouse focuses on rehosting these types of firmware services. To minimize the execution overhead, we rehost services using QEMU-user to emulate service binaries inside a `chroot` file system, which we term *single-service rehosting*, as opposed to full-system rehosting (via QEMU-system) that other solutions use. Single-service rehosting runs the target service binary in user-space and does not emulate any kernel modules.

4 Greenhouse Overview

Greenhouse is an automated system for single-service rehosting of single firmware services. It comprises three main components: the Runner, Checker, and Fixer. Supplementing these main components are an Extractor component that

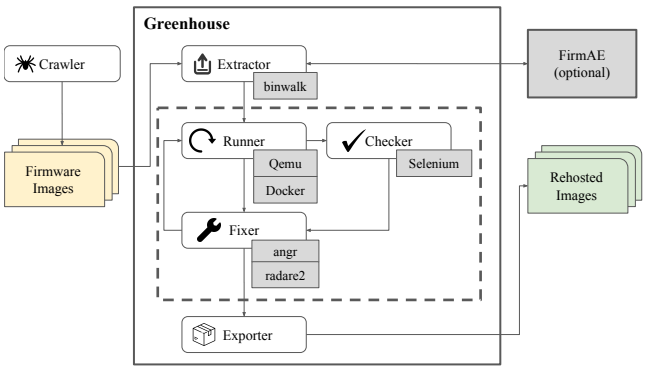

Figure 2: An overview of the Greenhouse pipeline.

performs the initial image extraction as well as an Exporter component that packages the rehosting results for later use. Outside of Greenhouse we created a Crawler module that built the initial firmware dataset used in our evaluation.

A firmware image being rehosted by Greenhouse starts at the Extractor, spends the majority of the rehosting process in an iterative loop among the Runner, Checker, and Fixer, and finally exits the process as a standalone Docker container via the Exporter. This process is fully automated.

While other works like Firmadyne [5], FirmAE [21] and Costin et al [9]. contain similar components that perform one or more of these steps, they apply all fixes at once. Greenhouse monitors the execution of a firmware service and selectively apply necessary fixes, which maximizes fixing opportunities while minimizes reduction to execution fidelity. To the best of our knowledge, ARI [26] is the only other work that iteratively rehosts.

4.1 File System Extraction

Like existing rehosting solutions [5,8,12,21,26,41], Greenhouse uses Binwalk to extract with all its optional dependencies installed. We run Binwalk with `-M` (Mashotrya) and `--preserve-symlinks` to recursively extract the root file system from the firmware image and preserve symlinks. Theoretically, Greenhouse supports rehosting *any* type of single service. In our evaluation, we rehost HTTP webservers, UPnP servers, and DNS servers.

4.2 Target Emulation

The Runner is the core of Greenhouse's iterative rehosting loop. It executes each web server inside a Docker container via QEMU-user. We use a Docker container to facilitate teardown and setup between iterations, because each run leaves artifacts in the file system and environment that may affect subsequent emulations if not cleaned up. We use `chroot` to ensure that the file system that is visible to the rehosted web server is the same as the on-device file system. Runner supports two tracing modes: In partial tracing mode, it collects syscall traces of the parent process and all child processes. In full tracing mode, it collects syscall and instruction traces (including addresses of all executed basic blocks) of the parent process and all child processes.

The Runner starts the web server, waits for up to 60 seconds, then invokes the Checker component to test the web server. The emulation and the rehosting loop terminate if the Checker component deems the service as successfully rehosted. Otherwise, the Runner parses the generated trace logs and waits for a potential wait loop. If it detects a wait loop or if the time spent in emulation exceeds a threshold the Runner forcibly terminates the emulation. Otherwise, it continues waiting and running the Checker against the web server periodically (e.g., every 10 seconds). This design is to handle the significant variance in start-up time between firmware services—using a fixed delay was infeasible.

At the end of emulation, all trace logs are collected and sent to the Fixer.

4.3 Fidelity Testing

The Checker component tests the fidelity of the rehosted service (which the Runner emulates) and passes the results to the Fixer, which then determines what interventions to apply.

The Checker takes as input the brand name of the firmware and an initial list of potential ports to test. We use a Checker that is specific to the service type (HTTP, UPnP, or DNS) to check for connectivity and test behaviors of the service. The Checker uses the result of checks and trace logs from the Runner to determine the level of Execution Fidelity of the rehosted service (detailed in Section 5).

While we only implement three types of Checkers for evaluation, users may plug-in other Checkers for rehosting other types of services.

4.4 Service Fixing

The Fixer performs run-time interventions that are used by Greenhouse to bypass rehosting roadblocks encountered during the iterative rehosting process. It uses traces and error logs from the Runner to diagnose potential roadblocks that limit Execution Fidelity. For each roadblock, the Fixer applies the corresponding intervention, which we will detail in Section 6.

After applying interventions for all identified roadblocks, the Fixer passes the modified file system and web server binary back to the Runner to initiate the next rehosting iteration. We repeat this loop until the emulated image is rehosted to a sufficient level of Execution Fidelity (as determined by the Checker), until we reach a point where we are unable to improve the fidelity, or we reach the maximum number of iterative cycles (empirically, 25 in our experiments). The rehosted file system is then packaged by the Exporter.

4.5 Exporting Results

The Exporter creates a tar file containing the rehosted file system, a set of scripts for running the rehosted service, meta-information such as username and passwords for logging in, and a Docker compose file that specifies container-level information (e.g., network devices) that is needed to run the rehosted service.

5 Rehosting Metrics

Greenhouse's iterative approach gradually improves the execution fidelity for user-space emulation until rehosting succeeds. This section discusses the metrics Greenhouse uses to determine the degree of success it achieves on a given service, broken into stages, and the reasoning behind how Greenhouse defines and measures the success in each stage. Each stage represents a tangible improvement to level of Execution Fidelity during emulation for the target service, as discussed in Section 2.2.

Existing rehosting techniques define a wide range of success metrics, depending on their goals and analysis focuses. These metrics include mapping execution traces to expected behaviors [20, 40], detecting functional network behaviors [5, 9, 21], or executing without crashing [41]. The ability to find new undiscovered vulnerabilities through successful analysis is taken as a further proof that rehosting is effective [7, 14, 28, 32, 40–42].

Separating each of these stages are *rehosting roadblocks* that hinder progression to the next level of Execution Fidelity. As discussed in Section 2.4, these roadblocks are discrepancies between the original device and the emulated environment where the rehosted service runs. We empirically determine a set of common roadblocks through manually examining hundreds of firmware samples and develop *interventions* for many of them. Section 6 discusses the types of roadblocks encountered and the interventions of Greenhouse to resolve them in greater detail.

By iteratively applying these interventions based on the roadblock encountered, Greenhouse can drive the specific Extraction Fidelity and Execution Fidelity of the service up to the level where dynamic analysis techniques (e.g., fuzzing) can be effectively applied.

Stage 1: Unpack. To begin single-service rehosting, Greenhouse must first unpack a firmware image and extract from it a complete file system. Because Greenhouse only supports Type-I firmware, we consider success at this stage to be the extraction of a recognizable Type-I Linux-based file system, which is indicated by the presence of a shell (e.g., `/bin/busybox` or `/bin/sh`) binary with a supported architecture. Failing to locate these binaries is an indicator of low Extraction Fidelity, which requires additional unpacking effort.

Stage 2: Execute. Rather than trying to emulate the entire boot environment, Greenhouse locates an executable binary that is associated with the target service using a list of common executable names. This list varies depending on the type of target service (HTTP, UPnP, or DNS). Greenhouse verifies if the identified binary can execute inside a `chroot` environment using QEMU-user. As this stage is more concerned with achieving high Extraction Fidelity than Execution Fidelity, we consider the stage successful even if the process immediately exits or crashes after execution.

Stage 3: Connect. The goal of this stage is to achieve a minimal level of communication with the emulated firmware service. This usually requires the emulated service to execute past its environment checks and bind to one or more ports at its desired addresses. Reaching this stage is crucial for any dynamic vulnerability analysis techniques, as many firmware exploits involve communicating with its network-facing services.

We consider this stage successful if we can connect to the rehosted service without it terminating, timing-out, or prematurely crashing. Becaue different services have different network protocols, each plugin has its own logic for checking connections. For example, the HTTP plugin sends an HTTP request and checks that a response is received (while ignoring the Status Code of the response), which is the same as the success metric in FirmAE. Success in this stage indicates that the rehosting has achieved a low level of Execution Fidelity, which may be sufficient for dynamic analysis in some cases (e.g., finding vulnerabilities in request parsing code of a web server).

Stage 4: Interact. Once the low level of Execution Fidelity is reached, Greenhouse attempts to drive the rehosting to as high a level of Execution Fidelity as possible. Note that Greenhouse may perform interventions that barely improve or even detract from other parts of the emulation but that specifically improve the fidelity of our target service for the purposes of our analysis. For example, Greenhouse may remove a CAPTCHA check for a service to streamline fuzzing for crashes inside the CGI handlers exposed by a web server.

Naturally, the Interact Checkers vary greatly between service types. To determine if a web server is running at a high Execution Fidelity level, Greenhouse performs basic interactions with the emulated web service. It checks the status codes of HTTP responses and compares the returned content against a set of pre-set error strings to identify malfunctioning backends. It also uses Selenium to load dynamic content and attempt some common login protocols. For DNS servers, the DNS plugin in Greenhouse makes a request to resolve *localhost* and parses the resulting DNS reply (if there is any). By developing protocol-specific Checkers, Greenhouse provides generalized heuristics for determining the fidelity of network-facing services without leaning into corner cases. While these checks are not an exhaustive test of the service being rehosted, our evaluation in Section 7 will show that rehosting a service to this stage is sufficient for many vulnerability discovery and assessment tasks.

6 Roadblocks and Interventions

When trying to improve the Extraction and Execution Fidelity of an image, multiple complications may arise that limit progress. We term these obstacles *rehosting roadblocks*, and their corresponding solutions *interventions*[1]. This section identifies the common roadblocks and presents several automatable interventions for them, which we implement in Greenhouse. It also discusses how these roadblocks might differ between single-service and full-system rehosting.

6.1 Roadblocks

While the exact set of complications differs from firmware to firmware, in the course of developing Greenhouse, we observed that there are many similarities and overlaps between them, even across different brands. Previous studies have discussed at length the issues of Missing Paths (R1), Peripheral Access (R3), NVRAM support (R4), and Network Interfaces (R5); We include them for completeness and to highlight how considerations from user-space rehosting might affect these roadblocks.

Missing Paths (R1). The initial extraction contains broken symlinks, missing files/folders, or missing/misplaced library files. Often, these files are generated or unpacked as part of the initialization scripts run at boot, and contain data critical for proper execution of firmware binaries. Full-system rehosting solutions run these initialization scripts as the default OS booting behavior, while single-service rehosting must infer these paths if the rehosted service does not generate them.

Runtime Arguments (R2). Some binaries take specific configurations on the command-line during runtime (e.g., path to webroot content, default starting port to bind to, etc.). This problem is unique to single-service rehosting, as most full-system rehosting solutions delegate the issue to the initialization scripts that the OS runs when booting.

Peripheral Access (R3). A common roadblock to both full-system and single-service emulations. Firmware services may try to communicate with hardware peripherals in a variety of ways ranging from accessing a file under `/dev` to directly accessing a reserved memory region. In our limited emulation these behaviors usually result in a crash or exit.

NVRAM Configurations (R4). NVRAM (Non-Volatile Random-Access Memory) is a hardware component common to many firmware routers. Previous works like FirmAE and Firmadyne have identified it as a core component that usually contains default configuration data necessary for the firmware image to start up and function. As QEMU does not explicitly implement NVRAM, both full-system and single-service emulations must address this.

Hard-coded Network Devices (R5). Networked services may have hardcoded "default" IP addresses or device names that they bind to. The service fails if a network device with that address or name is not present. This is further complicated in single-service rehosting as the TCP/IP stack implementation in QEMU-user is incomplete (e.g., it does not support `IPV6_RECVORIGDSTADDR`).

Multi-Binary Behavior (R6). Besides configuration files generated by initialization scripts at startup, some web servers may generate content or load configurations via IPC (interprocess communication) with separate, daemonized processes running in the background. Single-service rehosting must explicitly run these processes, while full-system rehosting does so as part of the OS boot process.

Environment Checks (R7). A catch-all category that covers any miscellaneous checks that the firmware binary might perform on its environment. Examples include: checking for DNS/web access, checking the user/group we are executing under, checking environment variables, and checking CPU resource usage. We characterize these checks by their common behavior of exiting if certain conditions are not met.

Environment Mangling (R8). Many firmware binaries run in an enclosed environment, which enables them to behave with complete disregard for other processes. These behaviors may mangle or corrupt parts of the emulation environment that are not intended to be visible to the emulated service. For example, a firmware binary may redirect stdout and stderr, then close every other file descriptor, under the assumption that it is the only process making use of file-system I/O. This mangles the logging done by QEMU-user, on which Greenhouse relies to gather critical data for identifying roadblocks. Full-system rehosting may avoid this problem by using OS-level logging infrastructure.

6.2 Interventions

Each of the aforementioned roadblocks has a corresponding intervention that we perform. Interventions either try to *fulfill* the criteria imposed by a Roadblock or bypass it. While fulfilling a Roadblock usually leads to better fidelity while bypassing lowers it, identifying the criteria of each specific Roadblock on each specific system is not a scalable solution.

Greenhouse implements a "best-effort" intervention system that tries to fulfill as many roadblocks as possible before falling back to the patcher.

Previous research described solutions to Roadblocks R1, R3, R4, and R5. Some solutions resemble the ones below, such as File Setup (I1), File Sanitization (I2), Boot-up Synchronization (I3), using a nvram-faker library (I4), and creating dummy Network Interfaces (I6). Unique to Greenhouse are the interventions that address user-space only roadblocks (I5, I7, and I8) and our *patching system* that manipulates the service binary to bypass checks. We also adapted several existing interventions to user-space due to complications from Environmental Mangling (R8).

File Setup (I1) - *[R1, R3, R7].* Using `strace`, we detect missing files by filtering for common file access system-calls such as `open()` and `access()`. We parse them for the ex-

[1] Prior work referred to interventions as *arbitrations* [21], *augmenting* [40], *mitigations* [26] and *refinement* [13].

pected paths and create a corresponding empty file or folder in the rehosted file system. If we can find a backup of the file, or if the file was misplaced in our environment, we copy it to the desired location. Unlike previous rehosting approaches, Greenhouse only adds missing files that the rehosted service tries to access. This minimizes conflicts with dynamically generated files during runtime.

File Sanitization (I2) - *[R3, R7].* We observed that peripheral access is sometimes performed through I/O operations on `/dev/` nodes, and can be bypassed by replacing the node with an empty file. This handles cases where a service appears to hang due to a blocking `read` call on a non-existent device interface, pipe, or socket. We thus "sanitize" the file-system immediately after unpacking when extraction is done by replacing all special files, except for symlinks (block, character device, pipe, and socket), with empty regular files. This approach is based on similar solutions used by FirmAE and Costin et al.

Boot-up Synchronization (I3) - *[R1, R2, R7].* We use FirmAE to run a full-system emulation of the image and obtain its "boot-up data," namely the runtime processes, files, and QEMU serial logs created or started by its boot-up and initialization scripts. Examples include files that are generated during boot, command-line arguments that are passed to the service, and any configuration data loaded into NVRAM. We use the boot-up data to improve the fidelity without simulating the OS boot process in user-space. Our approach differs from similar works [40,41] in that Greenhouse is augmented by but not dependent on FirmAE, as demonstrated by the set of firmware services that only Greenhouse could rehost in Section 7. We also perform an ablation study to show that Greenhouse performs comparatively well without this intervention.

nvram-faker (I4) - *[R4].* *nvram-faker* is an open source project that emulates the behavior of common NVRAM library functions by storing key-value pairs as files [10]. Firmadyne and FirmAE use similar libraries to tackle the same roadblock. We extend it with the *nvram-set* feature and provide wrappers for more NVRAM functions. We cross-compile a standalone version of the library and replace the default `libnvram.so` in the extracted file system. Our improved nvram-faker logs all keys used by the firmware, including ones that are not part of its current dictionary. Greenhouse then tries to provide a key-value pair in the next iteration of the emulation based on the NVRAM values from boot-up data or from a list of default pairs. As observed by FirmAE, providing an empty string value significantly reduces crashes. Greenhouse takes this further by using a dictionary of common key-value pairs for each brand generated from online sources and our dataset. During the evaluation, we found that providing these non-empty default values transforms many partially rehosted services to fully rehosted.

Runtime ArgParser (I5) - *[R2].* We provide a fallback intervention if command-line arguments for the firmware binary cannot be obtained from boot-up data. A simple heuristics-based regex parser uses the brand and name of the web server to parse for potential runtime arguments and supply common defaults to them. Both single-service and full-system rehosting may conduct this intervention.

Dummy Network Devices (I6) - *[R5].* We adapt this approach from FirmAE. In Greenhouse, a modified version of the QEMU `bind()` syscall logs whenever a bind is attempted on a network address or device name. Greenhouse then configures Docker containers in our emulation with the corresponding network bridge. If the binary must run outside Docker containers, Greenhouse provides an automatically generated script that creates the dummy devices.

Background Script Plugins (I7) - *[R6].* Although a generic solution to handle *multi-service* targets is out of scope, Greenhouse provides a specific solution for target services that must communicate with other background processes. Greenhouse allows pluggable heuristics for identifying executables for background processes and how to run them. Greenhouse provides plugins for `xmldb` and `config`, which are used by D-Link and Netgear firmware respectively to set and retrieve configuration data.

IPv6 Workaround (I8) - *[R5].* Many new IoT devices, routers included, use IPv6 for communication. This is not always supported by the host machine where Greenhouse runs, such as the Kubernetes cluster where we conduct our evaluation. The implementation of IPv6 in QEMU-user is also incomplete. Hence, we implemented a workaround in QEMU to divert all binding to IPv6 addresses to the IPv4 address 0.0.0.0, and ensure socket operations with unimplemented or unsupported IPV6 flags always succeed.

Patching `sysinfo()` (I9) - *[R7].* Some routers adaptively disable services based on load, which makes dynamic analysis difficult. This was particularly notable during large-scale analysis when running multiple Greenhouse emulations on the same Kubernetes node. We patch QEMU so `sysinfo()` always returns 0 to prevent these behaviors.

Logging Behavior (I10) - *[R8].* Our emulation in user-mode shares file descriptors with the emulated service. This problem is exclusive to single-service rehosting. To ensure that our emulation trace logs are not mangled, we modify the `open()` and `close()` syscalls, as well as the behavior of QEMU's `trace` function, to reserve a range of file descriptors (300–400) for our log files. Attempts to close descriptors in this range will return failure.

6.3 Patching

In the case that our specific interventions are insufficient, Greenhouse attempts to directly patch the firmware binary to bypass the section of code that prevents it from reaching the next stage. While this might potentially lower the Extraction and Execution Fidelity of the firmware binary, we found that by limiting the type of patching done to specific conditions, it is surprisingly effective at enabling further rehosting.

Greenhouse handles three types of patches: a Premature Exit patch, a Wait Loop patch, and a Crashing Instruction patch. To identify the relevant patch automatically, we use *angr* [30] to construct a context-sensitive control-flow graph (CFG) of the firmware binary. Each node in the CFG is differentiated by its address and the addresses of all the blocks that call it in the CFG. This allows us to map the execution trace of a binary to the CFG for Premature Exit and Wait Loop patches.

Premature Exit Patch. The Premature Exit Patch handles the general case where the binary tests the result of some manner of check, then branches to an exit function if the check fails. By identifying the branch instruction taken by the firmware binary that leads to the exit function, we can flip the branch to bypass the check altogether. To do so, Greenhouse maps the execution trace of the firmware binary to the context-sensitive CFG and scans for call to `exit()` or `abort()`. In case no such function signature can be found, but the binary exited cleanly, it assumes that the last instruction in the execution trace is the exit function.

The Patcher then recursively prunes the context sensitive CFG from the exit call to the nearest dominator within the mapped trace that is the parent of a child that it (1) does not dominate and (2) is not part of the original trace. Because all nodes prior to the dominator eventually lead to the exit, we may assume that this block corresponds to the critical branch point that leads to the exit. We then search the block for the relevant jump instruction and patch it to point at the untaken branch.

In practice, this Patch is effective at handling most Environment Checks (**R7**) and certain types of Peripheral Access (**R3**). It can also help compensate for interventions that result in empty file reads (**I1**) when the binary expects content.

Wait Loop Patch. The Wait Loop Patch handles cases where instead of exiting the binary is instead trapped in a constant loop waiting for external input from another process. Examples include a `poll()` for network activity in a basic web connectivity check, or a `sleep()` loop while waiting for a particular peripheral to connect.

The Wait Loop Patch uses the same context-sensitive CFG mapped to the execution trace to determine that a program is looping. It attempts to find a branching node that is not part of the original execution trace and does not immediately lead back to the loop. As many firmware binaries are essentially large loops, we constrain ourselves to "tight" loops of no larger than 30 basic blocks.

To ensure we do not inadvertently patch the section of code responsible for handling incoming server requests, this patch is only invoked in cases where no network connectivity was detected despite timing out. Similar to the Premature Exit Patch, the Wait Loop Patch is a generic intervention that handles a subset of Environment Checks (**R7**) and Peripheral Access (**R3**) roadblocks.

Crashing Instruction Patch. The Crashing Instruction Patch usually does not need the CFG. It takes the address of the last recorded instruction in the execution trace, maps it to its respective basic block in the firmware binary, and patches the next instruction that would be executed, replacing it with `nop`s. In the case where the instruction is outside the address space of the binary, such as inside a library call, the Patch uses the CFG to determine the address of the caller instruction and place a `nop` there instead.

This patch is only invoked when a segmentation fault is detected during emulation, as it assumes that the next instruction that would have been recorded caused the fault. While this can be highly destructive to the firmware binary, and the overall fidelity of the emulation, it is one of the cleanest ways to handle invalid direct memory accesses such as in **R3**.

7 Evaluation

We design a series of experiments to answer the following research questions:

- How does Greenhouse's rehosting performance compare to state-of-the-art full-system rehosting solutions? (Section 7.2)
- What factors impact the rehosting performance of Greenhouse? (Section 7.3)
- Does Greenhouse reach a level of Execution Fidelity that enables vulnerability discovery and risk assessment? (Section 7.4)
- How much does Greenhouse-rehosted service improve the fuzzing performance? (Section 7.5)

Evaluation Environment. We conducted all experiments on a Kubernetes cluster that contains 42 nodes and over 2,000 CPU cores. We ran 300 pods in parallel and assigned each pod a minimum of 2 CPU cores and 16GB of RAM. We modified FirmAE and EQUAFL to run on a Kubernetes cluster. Our modifications will be released when we open source all research artifacts.

7.1 Firmware Image Collection

To ensure we have a wide coverage of router models with the most up-to-date firmware samples, we built our own firmware image collection by crawling websites of nine well-known router brands (ASUS, Belkin, D-Link, Linksys, Netgear, Tenda, TP-Link, TRENDnet, and Zyxel) and downloading all versions of router and camera firmware images that are available. This provided 12,943 firmware images. We filtered them and removed encrypted or incomplete images and any images that do not resemble Type-I Linux-based firmware. We also removed images that ran on unsupported architecture. Then, we merged the remaining images with FirmAE's data set and removed duplicates. As with FirmAE, we obtained the latest version of firmware for each device (as of February 2023), identified by model, revison number, and region. Our final collection of firmware images includes 7,140 unique

		FirmAE				EQUAFL				Greenhouse			
Brand	**Initial**	**Unpack**	**Execute**	**Connect**	**Interact**	**Unpack**	**Execute**	**Connect**	**Interact**	**Unpack**	**Execute**	**Connect**	**Interact**
ASUS	846	843	843	568	11	801	241	14	0	829	822	791	789
Belkin	63	63	57	31	6	60	8	2	2	61	56	46	33
D-Link	1,426	1,384	1,071	734	515	1,099	493	200	190	843	721	568	512
Linksys	92	84	81	61	40	85	13	7	4	83	34	24	23
Netgear	2,712	2,606	2,499	1,632	1,129	2,334	706	145	56	2,273	2,010	1,517	1,094
Tenda	173	161	158	35	10	148	13	0	0	156	103	71	64
TP-Link	1,064	1,052	994	572	441	1,016	636	149	145	902	428	170	66
TRENDnet	744	718	674	415	240	692	300	52	43	523	463	332	254
Zyxel	20	20	20	10	6	20	7	2	2	20	15	8	6
Total	7,140	6,931	6,397	4,058	**2,403**	6,255	2,417	571	**442**	5,690	4,652	3,525	**2,841**

Table 2: Numbers of FirmAE-, EQUAFL- and Greenhouse-rehosted web servers that reached each of the four rehosting milestones, organized by brand.

images across 1,764 unique devices. This collection is 6.3x FirmAE's data set (with 1,124 images) and 3.7x the dataset from Costin et al (with 1,925 images).

7.2 Firmware Rehosting Results

We compare Greenhouse against FirmAE and EQUAFL. EQUAFL, as a fuzzing solution, is an extension of Firmadyne. Because FirmAE is based on and performs strictly better than Firmadyne, we do not evaluate against Firmadyne.

We also considered Costin et al., which examines multiple emulation approaches and settles on full-system emulation via QEMU. Because it is similar to Firmadyne, we do not compare Greenhouse against Costin.

Different types of services have different network protocols. Our rehosting platform currently supports three types of networking services: HTTP, UPnP, and DNS. We focus our evaluation on web servers as prior research did, while reporting rehosting results for UPnP and DNS servers.

Determining levels of Execution Fidelity. We use the Checker component to determine the Execution Fidelity in terms of the stages described in Section 5. Additionally, we parse the logs of Greenhouse, EQUAFL, and FirmAE to determine the degree to which Unpack and Execute succeeded for each image. For Greenhouse, we consider Unpack successful if we find a web server executable, and a successful Execute if we can run it in QEMU-user until a `bind()` syscall is detected. For EQUAFL and FirmAE, we consider Unpack successful if they can find and mount a file system image, and successful Execute if they can boot the image in QEMU-system to the point that they attempt to enable network interfaces. We define a successful Connect for FirmAE to be a curl request that does not time out per their implementation.

Results. Table 2 shows the numbers of rehosted firmware services (or images for FirmAE) that reached each rehosting stage. Table 3 shows the number of successful rehosts for each device for the latest firmware in the dataset. Overall, the number of Greenhouse-rehosted firmware services is comparable to that of FirmAE for both latest (538 vs. 558) and total (2,841 vs. 2,403) firmware images. Greenhouse is significantly more successful for some brands (e.g., ASUS, Belkin, and Tenda) while less successful for other brands (e.g., Netgear and TP-Link). EQUAFL could execute 2,417 targets but only rehost 442, and failed to rehost two brands (ASUS and Tenda).

	Devices	**EQUAFL**	**FirmAE**	**Greenhouse**
asus	157	0	2	138
belkin	63	2	6	33
dlink	355	35	100	101
linksys	73	2	30	18
netgear	311	11	131	118
tenda	105	0	5	30
tplink	489	76	217	28
trendnet	191	10	61	66
ZyXEL	20	2	6	6
TOTAL	1,764	138	558	538

Table 3: Number of successfully rehosted HTTP web-services (that reach Interact) for the latest version of each firmware device in our dataset.

	FirmAE only	**Greenhouse only**	**Intersection**	**Union**
ASUS	9	787	2	798
Belkin	2	29	4	35
D-Link	80	76	436	592
Linksys	30	12	11	53
Netgear	495	457	637	1,589
Tenda	1	55	9	65
TP-Link	422	47	19	488
TRENDnet	98	112	142	352
Zyxel	3	3	3	9
Total	1,140	1,578	1,263	3,981

Table 4: Overlaps of firmware services that FirmAE and Greenhouse successfully rehosted (i.e., reaching the Interact milestone), as well as numbers of services that are only rehosted by one solution, organized by brand.

Overlaps between rehosted services. We examined the overlap between FirmAE- and Greenhouse-rehosted services. We exclude EQUAFL due to its poor rehosting performance. Interestingly, as shown in Table 4, the set of firmware services that Greenhouse fully rehosted has little overlap with the ones that FirmAE rehosted. Together, FirmAE and Greenhouse can rehost 3,981 out of 7,140 services, which cover nearly 50% more than either solution can individually rehost. This shows that Greenhouse handles unique rehosting obstacles that full-system emulation techniques cannot.

Rehosting other services. We used SaTC [6] to first identify

the names of executables for common networked services. We manually curated these names to generate lists of common HTTP, UPnP, and DNS server binaries associated with these services. Table 5 shows that Greenhouse rehosts 50.1% of found HTTP web servers, 43.9% of found UPnP servers, and 47.2% of found DNS servers. This demonstrates that Greenhouse's approach is not limited to web servers, and can extend to rehost other types of firmware services.

	HTTP		UPnP		DNS	
	Found	**Interact**	**Found**	**Interact**	**Found**	**Interact**
ASUS	829	789	824	311	781	635
Belkin	61	33	47	4	41	24
D-Link	844	512	456	90	475	77
Linksys	83	23	42	2	68	54
Netgear	2,272	1,094	2,004	1,203	1,534	636
Tenda	156	64	108	71	49	21
TP-Link	902	66	430	83	256	33
TRENDnet	521	254	226	57	278	163
ZyXEL	7	6	15	1	13	7
Total	5,675	2,841	4,152	1,822	3,495	1,650

Table 5: Number of services of type (HTTP, UPnP, DNS) for which Greenhouse was able to locate a web server for, and subset which we successfully rehosted.

Impact of each intervention. To demonstrate how each intervention contributes to the overall rehosting successes of Greenhouse, we rerun Greenhouse to rehost HTTP web server binaries multiple times with one of the eight interventions (I1–I8) disabled each time. We do not include Interventions I9 and I10 because they are necessary for the operation of Greenhouse. We also include a run with the Patcher (Section 6.3) disabled to study the impact of our binary patching component. Table 6 shows the breakdown of each run with a particular intervention disabled.

Notably, Greenhouse can rehost 2,455 (86.4%) of the 2,841 targets in the full run without augmenting its boot-up environment with data from FirmAE (no_bootsync). This number drops significantly (to 1,787 or 62.9%) when our heuristics-based runtime argument intervention (no_args) is disabled. We also note that disabling IPv6 workarounds (no_ipv6) has minimal impact on the rehosting result (2,562 or 90.2%). After a manual investigation, we found that in many cases, the firmware service supports both IPv4 and IPv6, and the patcher forced the service to execute with only IPv4 without termination. Running without file sanitization (no_sanitize) caused significant issues in our large-scale pipeline: Not replacing special files caused many firmware samples to hang, crash, or even corrupt the rehosting environment. Therefore, the numbers in the Table 6 for I2 (marked with *) are an extrapolation based on the samples that did finish.

7.3 Case Studies

7.3.1 ASUS Firmware

We examined ASUS firmware services for which Greenhouse rehosted 789 out of 846 to the Interact stage while

Intervention Disabled	Unpack	Execute	Connect	Interact
no_setup (I1)	4,886	2,523	1,412	884
no_sanitize (I2)*	1,477	990	923	895
no_bootsync (I3)	5,696	4,222	3,153	2,455
no_nvramfaker (I4)	4,309	3,272	1,714	1,075
no_argparser (I5)	5,524	3,507	2,034	1,787
no_dummy (I6)	5,591	4,627	3,474	2,690
no_bgscripts (I7)	5,613	4,562	3,295	2,438
no_ipv6 (I8)	5,614	3,671	3,293	2,562
no_patcher	5,617	4,273	3,177	2,644

Table 6: Impact of Greenhouse Interventions I1-I8 and the Patcher on rehosting successes of HTTP web servers, as discussed in Sections 6 and 7. Each row represents a Greenhouse large-scale rehosting attempt on our dataset of 7,140 firmware images with the corresponding Intervention disabled.

FirmAE only rehosted 11. While both Greenhouse and FirmAE rehosted similar numbers of services to the Execute stage, FirmAE could not Connect to almost half of them, and only 11 were able to Interact.

Configuration Reuse. We first analyzed 257 ASUS services that FirmAE rehosted to Execute but was unable to Connect, while Greenhouse successfully rehosted to Connect. This represents the set of services where our interventions likely mitigated roadblocks that are related to network connectivity. Greenhouse made use of NVRAM data from external sources for 255 of them, most notably NVRAM data from other Netgear images. Critical NVRAM values include `lan_ipaddr` and `env_path` that directly affect service execution. By reusing configuration information from other images in our collection, Greenhouse rehosted more services to the Connect stage.

Iterative Interventions. We then analyzed 531 ASUS services that FirmAE rehosted until Connect but not Interact, and that Greenhouse rehosted to Interact. For most of these services (461 out of 531), the emulated service in FirmAE returned an HTTP status code of 200, but the actual web pages displayed file-not-found error messages [2]. The remaining services either timed out, returned an empty HTML page, or experienced authentication issues. Such services are less suitable for analysis as they are likely missing parts of the emulation environment that impact the service of interest.

In 517 of these cases, Greenhouse addresses the issue through iterative interventions, including relocating missing files (e.g., `QIS_wizard.html` or `cert.pem`). This shows the impact of interventions of Greenhouse on the Execution Fidelity of rehosted services.

In summary, Greenhouse outperformed FirmAE due to interventions and the iterative application of them. We also migrated critical configuration data across firmware that FirmAE does not.

[2] According to RFC 2616 [1] an HTTP 200 status code means "the request has succeeded," yet clearly these devices do not follow the specification.

7.3.2 Tenda Firmware

Greenhouse used the Wait Loop Patcher to patch 52 out of 55 Greenhouse-only rehosted Tenda services and applied the Premature Exit Patcher on 26 out of 52 Tenda services. After manual analysis, we identified a key roadblock on some Tenda services: the `ConnectCFM()` function. This function accesses the `cfm` binary, which interfaces with the CFM peripheral that persists configuration data across reboots [37]. When unable to access CFM, web servers either retry infinitely or exit with the error message "connect cfm failed!" before initiating any network behaviors. By patching the check leading the code to exit or loop, Greenhouse forced the execution into network-facing code.

7.3.3 TP-Link Firmware

We examined why Greenhouse successfully rehosted much fewer Netgear and TP-Link services than FirmAE. The major reason is that many web servers in Netgear and TP-Link firmware heavily relied on communication with other processes that the web servers themselves did not start. For example, a TP-Link web server proxies all HTTP traffic to another service through dbus-daemon, and both dbus-daemon and the other service must be started by init.d scripts. Strictly speaking, these targets do not belong to single-service rehosting, but we still report them as rehosting failures for fairness. We leave single-user, multi-service rehosting to future work.

7.4 Vulnerability Risk Assessment

To evaluate the applicability of rehosted services for vulnerability risk assessment, we follow FirmAE's convention and use the automatic exploit framework, RouterSploit. Previous works also used RouterSploit in evaluation [5, 9]. We selected all rehosted web servers that reached at least Connect for Greenhouse (3,526) and FirmAE (4,058) and replayed 125 known N-day exploits against each service.

	FirmAE				Greenhouse			
	PD	**CI**	**IL**	**AB**	**PD**	**CI**	**IL**	**AB**
ASUS	0	16	0	0	0	0	0	0
Belkin	3	2	0	3	1	17	0	0
D-Link	168	351	149	21	97	281	84	10
Linksys	0	1	0	0	0	0	0	0
Netgear	3	79	0	0	60	79	60	0
Tenda	0	0	0	0	0	0	0	0
TP-Link	0	0	0	0	0	0	0	0
TRENDnet	13	11	24	0	5	17	5	1
ZyXEL	0	0	0	0	0	0	0	0
Total	187	460	173	24	163	394	149	11

Table 7: Breakdown of running 125 N-day exploits on rehosted services using RouterSploit. Each cell shows the number of exploits of a given type that successfully exploit a rehosted image. PD = Password Disclosure. CI = Command Injection. IL = Info Leak. AB = Authentication Bypass.

Results. As Table 7 shows, RouterSploit exploited 717 known vulnerabilities across 3,526 Greenhouse-rehosted firmware services. Meanwhile, RouterSploit found 844 known exploits on 4,058 samples rehosted by FirmAE. Despite not conducting full-system emulation or modeling peripherals, Greenhouse-rehosted services are sufficient to use for vulnerability risk assessment.

7.5 Fuzzing Rehosted Services

A key application for rehosting is automated vulnerability discovery, particularly fuzzing. To evaluate the applicability of Greenhouse-rehosted services for fuzzing, we use AFL++ to fuzz 3,526 firmware images that Greenhouse rehosted to a fidelity level of Connect. We compare our results to the closest work that does user-space fuzzing of Type-I firmwares, EQUAFL. Due to resource constraints, we limit our selection to 2,612 randomly chosen samples, making sure to include the 70 samples from the EQUAFL dataset, even if Greenhouse could not rehost all of them. Because the full EQUAFL dataset is not available at the time of writing, we use the latest firmware image of a device when we cannot locate the original sample that EQUAFL used.

Additionally, we conduct a performance evaluation of Greenhouse's user-space emulation on eight firmware images from the EQUAFL dataset using Greenhouse+AFL and EQUAFL. As EQUAFL positions itself as a strict improvement over FirmAFL, we do not reevaluate FirmAFL. Finally, we manually analyze the fuzzing results of eight rehosted EQUAFL images and six randomly-chosen images (Table 9) to confirm our rehosted emulation can find real-world vulnerabilities.

We chose AFL++ as the fuzzer and built a generic fuzzing harness for web servers, which emulates client connections in AFL-QEMU. We discuss the implementation of harness as well as why FirmAFL and EQUAFL do not generalize to unseen firmware targets in Appendix.

7.5.1 Performance Evaluation vs EQUAFL

Each fuzzing experiment ran inside a Docker container on a bare-metal server running Ubuntu 22.04 LTS with an 80-core Intel Xeon Gold 5218R 2.10GHz CPU and 270GB of RAM. For the eight EQUAFL targets, we ran our fuzzer and EQUAFL's fuzzer ten times for 24 hours each and measured both the execution speed and total corpus count over time.

Figure 3 shows the fuzzing performance using AFL on Greenhouse against EQUAFL for the eight rehosted `httpd` servers. On average, Greenhouse-rehosted services (native user-space emulation) were 2x faster than EQUAFL's synchronized filesystem approach.

7.5.2 Large-Scale Fuzzing

EQUAFL defines a fuzzable target as one where "the applicaton process can be initiated by the fuzzer without reporting errors explicitly." We use this same definition on the 2,612 fuzzed samples. Table 8 shows the fuzzability of the 2,612 samples rehosted by Greenhouse using a similar metric to

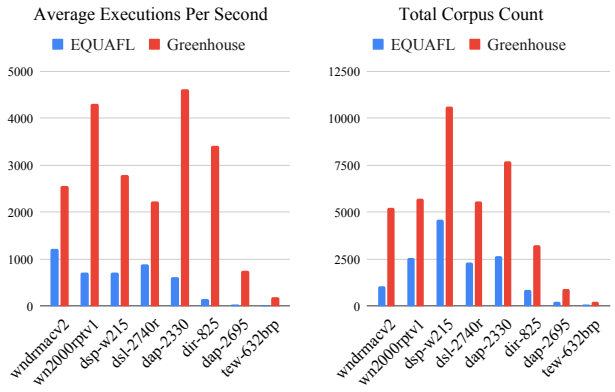

Figure 3: Fuzzing Performance of Greenhouse+AFL versus EQUAFL.

	Selected	SUCC	ERR	HAN	CRA
ASUS	502	451	18	22	11
Belkin	0	0	0	0	0
D-Link	409	290	19	74	26
Linksys	23	23	0	0	0
Netgear	1,462	957	324	0	181
Tenda	71	62	2	0	7
TP-Link	145	4	97	0	44
TRENDnet	0	0	41	0	0
ZyXEL	0	0	0	0	0
Total	2,612	1,787	460	96	269

Table 8: Fuzzability of Greenhouse rehosted and partially rehosted firmware images, where SUCC indicates a successful fuzzing run, ERR indicates that fuzzing started but encountered errors, HAN means the target hung, and CRA means the target crashed without starting fuzzing.

EQUAFL's evaluation.

We also found that of the 70 images that were part of the EQUAFL dataset, Greenhouse could only fuzz 45 of them. However, we note that Greenhouse could fuzz a total of 1,787 images (68.4% of the tested set) and find a total of 18,599 raw crashes across 733 targets. As the number of targets that EQUAFL can fuzz is limited by what its coupled full-system emulation can rehost, we expect EQUAFL be limited to no more than 571 targets. This demonstrates the scalability of Greenhouse's approach for large-scale rehosting and analysis.

7.5.3 Real-world Vulnerabilities

Due to the amount of effort needed to manually triage crashes for all 1,787 firmware images fuzzed, we limit our analysis to the subset of crashes discovered for the 14 services in Table 9. We first filter crashing inputs using `tmin` and `md5sum` to identify unique crashes. Then a human analyst examined these filtered crashes to see if they are false positives. In total, we found 79 unique crashes across these 14 services. We confirmed that 26 out of 79 are legitimate 0-day vulnerabilities in firmware services, and have disclosed this information to their respective vendors. None of the

	Tmin inputs (total crashes)	Unique crashes (md5 + manual)	Unique vulns
TEW_652BRP_v2.0R_2.00	9	1	1
AC1450_V1.0.0.6_1.0.3	23	5	1
FW_RT_AC750_30043808497	18	12	1
TEW-632BRPA1_FW1.10B31	42	11	6
DAP-2330_REVA_1.01RC014	14	1	1
WNDRMACv2_Version_1.0.0.4	23	7	0
DAP_1513_REVA_1.01	28	6	4
FW_RT_G32_C1_5002b	37	10	3
DAP-2695_REVA_1.11.RC044	22	1	1
WN2000RPT_V1.0.1.20	21	1	1
DIR-601_REVA_1.02	32	2	2
DIR-825_REVB_2.03	52	13	4
DIR-825 209EUb09	37	8	2
DSP-W215_REVB_v2.23B02	0	0	0
Total	**358**	**79**	**26**

Table 9: Vulnerabilities found through fuzzing 14 firmware images rehosted with Greenhouse+AFL.

79 crashes analyzed were introduced by binary patching in Greenhouse. A breakdown mapping each of these crashes to their respective vulnerabilities can be found in the Appendix.

8 Limitations

Encrypted firmware. Greenhouse needs a high Extraction Fidelity to perform its iterative rehosting approach. Our patching approach also uses embedded function symbols like `exit()`. We limit ourselves to rehosting firmware image that are relatively complete, cooperative (no hidden, malicious, or obfuscated binaries), and unencrypted.

QEMU limitations. Greenhouse uses QEMU-user to perform user-space emulation and is subject to any flaws and limitations of the emulator. For example, QEMU-user has limited support for the `clone()` syscall under MIPS. Due to limitations in the abstraction when running the emulator in user mode, unless `clone()` is called with very specific flags, QEMU will not emulate it properly. This results in a number of MIPSEL binaries that implement fork servers using `clone()` to fail. We estimate that this affects 147 out of 7,140 cases, about 2% of our firmware sample collection. We plan to fix this issue in QEMU-user and submit a patch upstream.

angr limitations. Greenhouse uses angr to create CFGs, and is thus subject to bugs and limitations in angr. A number of MIPS targets (68, or 0.9% of our collection) crashed due to assertion failures in angr when building CFGs.

Missing library functions. Some firmware makes use of libraries containing custom functions, usually for interacting with peripherals. If these libraries are missing during extraction due to low Extraction Fidelity, rehosting further is close to impossible. In some cases, these functions are found in the `libnvram` library that we replace, which further complicates the issue. Mitigating this roadblock would require dynamically injecting custom stub functions for each special case during rehosting, which we plan to address as an engineering problem in the future.

False positives from patching. Greenhouse's patching may

introduce false positives during fuzzing. Although we did not encounter any such false positives in our manually examined set, the possibility exists. The most likely way this can occur is if Greenhouse patches a branching instruction responsible for a security check. To mitigate this, Greenhouse keeps an original copy of the binary it patches inside the exported emulation, and a record of all instruction addresses patched. A human analyst who is triaging a crash can compare against the original binary to determine if the root cause is related to a Greenhouse patch.

9 Related Work

Large-scale emulation. Costin et al. [9] is one of the earliest works that examined the feasibility of different emulation approaches for firmware rehosting. It attempted rehosting via chroot and user-space emulation. However, the authors concluded that the fidelity loss was too much for user-space emulation to be useful, and decided to use a full-system emulator for rehosting. Recent works that perform large-scale rehosting of Type-I firmware [5, 21, 23, 38] also build upon and instrument OS-level emulators to automatically rehost firmware images. Like Greenhouse, these approaches automatically identify full-system emulation roadblocks and apply interventions to resolve them.

Greenhouse differs from these works by being more selective in what to emulate. Because it focuses on rehosting a single-service in user-space, Greenhouse can precisely and iteratively apply interventions. If no appropriate intervention is found, Greenhouse attempts to patch the binary.

ARI [26] takes a similar approach: It extends Firmadyne and applies interventions to handle different failure cases on Linux-based firmware. ARI iteratively emulates, tests, and fixes the firmware image using their own fidelity criterion, with a similar distinction between connectivity and interactivity as Greenhouse. However, ARI does not patch and still relies on Firmadyne's full-system emulation to rehost firmware images. Thus, ARI and Greenhouse are orthogonal techniques with some similarities.

User-space emulation. Researchers have explored fuzzing firmware images in user-space. FirmAFL [40] hybridizes user-space execution with full-system emulation. EQUAFL [41] fuzzes entirely in user-space by transplanting the filesystem state of a rehosted firmware image to user-space. As discussed in Appendix, these techniques require significant manual effort to support new targets and do not scale. Their performance also depends on the full-system emulation technique with which they are coupled. Greenhouse emulates a firmware service entirely in user-space, while being scalable to a much larger set of firmware images and analysis tools.

Other analysis approaches. Beyond fuzzing, researchers have achieved success via other techniques. Costin et al. [8] performed static analysis on firmware images by combining a correlation engine with simple keyword searches. Recent research analyzes programs by symbolically executing them to find control flow bugs [11, 19, 29] or statically analyzing the interactions between multiple binaries [6, 12] to detect insecure data flows.

Rehosting Type-II and Type-III firmware. Heapster [16] identifies the heap library used by the image for memory allocation and uses symbolic execution to detect potential vulnerabilities. PRETENDER [17] models MMIO behavior by tracing behavior on the actual device, while P2IM [14] tries to exhaustively probe for MMIO to generate a model. μEmu [42] uses symbolic execution to model the image and infer peripheral behavior from its constraints. Fuzzware [28] combines these approaches by implementing its own instruction set architecture emulator. It iteratively probes MMIO behavior via fuzzing and feeds the results into a symbolic execution engine to derive models that are used to update the emulation. Srinivasan et al. [31] tries to rehost Type-II images by repackaging them as Linux applications rather than complete firmware images.

The approaches taken by these works have a conceptual similarity to Greenhouse in how they take "slices" of the firmware image to emulate and expand their knowledge base from there. Future work could look to adapt techniques from one approach to the other.

10 Conclusion

We present Greenhouse, an automated system for large-scale single-service rehosting of Linux-based firmware in user-space. Greenhouse makes use of "best-effort" mitigations to iteratively adapt a firmware image and the emulated environment to each other. We also define a more stringent set of criteria for rehosting with respects to the end-goals of emulating for dynamic analysis and vulnerability discovery. We evaluate Greenhouse on a set of 7,140 Type-I firmware images and rehost 2,841 of them to the minimal level of usability under our new criteria. Using existing analysis tools like RouterSploit and AFL, we find 717 N-day exploits and 26 0-day vulnerabilities. This demonstrates both the feasibility of single-service, user-space emulation in creating usable emulated images for dynamic analysis.

11 Acknowledgement

The authors would like to thank the shepherd and anonymous reviewers for their guidance and feedback. This project has received funding from the following sources: Defense Advanced Research Projects Agency (DARPA) Contracts No. HR001118C0060, FA875019C0003, and N6600120C4020; the Department of the Interior Grant No. D22AP00145-00; and National Science Foundation (NSF) Awards No. 2146568 and 2232915.

References

[1] Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[2] Amazon Web Services. FreeRTOS. https://www.freertos.org.

[3] Arduino. Arduino – Home. https://www.arduino.cc.

[4] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

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

[6] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *USENIX Security Symposium*, pages 303–319, 2021.

[7] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security)*, pages 1201–1218, 2020.

[8] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security)*, pages 95–110, 2014.

[9] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448, 2016.

[10] Zachary Cutlip and Decidedly Gray. nvram-faker. https://github.com/zcutlip/nvram-faker. (Accessed on 2022-11-08).

[11] Drew Davidson, Benjamin Moench, Somesh Jha, and Ristenpar Thomas. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium (USENIX)*, pages 463–478, 2013.

[12] Redini et al. KARONTE: Detecting insecure multi-binary interactions in embedded firmware. In *IEEE Symposium on Security and Privacy (SP)*, 2020.

[13] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling security analyses of embedded systems via rehosting. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2019.

[14] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security)*, pages 1237–1254, 2020.

[15] The Eclipse Foundation. 2020 IoT developer survey key findings. https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2020.pdf. (Accessed on 2022-10-11).

[16] Fabio Gritti, Fabio Pagani, Ilya Grishchenko, Lukas Dresel, Nilo Redini, Christopher Kruegel, and Giovanni Vigna. HEAPSTER: Analyzing the security of dynamic allocators for monolithic firmware images. In *IEEE Symposium on Security and Privacy (SP)*, pages 1559–1559. IEEE Computer Society, 2022.

[17] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 135–150, 2019.

[18] Mohammad Hasan. IoT analytics: State of IoT 2022. https://iot-analytics.com/number-connected-iot-devices/. (Accessed on 2022-10-08).

[19] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *ACM SIGSAC Conference on Computer and Communications Security (ACMCCS)*, 2017.

[20] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security)*, pages 321–338, 2021.

[21] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic

analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[22] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[23] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar^2: A multi-target orchestration platform. In *Proceedings of the Workshop on Binary Analysis Research (Colocated with NDSS Symposium 2018)*, volume 18, pages 1–11, 2018.

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

[25] NIST. National vulnerability database. https://nvd.nist.gov/. (Accessed on 2022-10-08).

[26] Ryan William Ramseyer. *Automated Rehosting and Instrumentation of Embedded Firmware*. PhD thesis, Massachusetts Institute of Technology, 2021.

[27] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *USENIX Security Symposium (USENIX)*, 2020.

[28] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *USENIX Security Symposium (USENIX)*, 2022.

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

[30] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[31] Jayashree Srinivasan, Sai Ritvik Tanksalkar, Paschal C Amusuo, James C Davis, and Aravind Machiry. Towards rehosting embedded applications as Linux applications. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[32] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.

[33] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *USENIX Security Symposium (USENIX)*, 2018.

[34] Team82. State of XIoT security. https://claroty.com/resources/reports/state-of-xiot-security-1h-2022, 2022.

[35] Tenda. Tenda - Ac21 firmware CVE - OpenCVE. https://www.opencve.io/cve?vendor=tenda&product=ac21_firmware. (Accessed on 2023-19-05).

[36] threat9. routersploit: Exploitation framework for embedded devices. https://github.com/threat9/routersploit. (Accessed on 2022-10-08).

[37] Anton Viktorov. Tenda reverse. https://github.com/latonita/tenda-reverse. (Accessed on 2022-10-08).

[38] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *The Network and Distributed System Security (NDSS) Symposium*, volume 14, pages 1–16, 2014.

[39] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. (Accessed on 2022-10-08).

[40] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *USENIX Security Symposium (USENIX)*, 2019.

[41] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. 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*. ACM, 2022.

[42] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *USENIX Security Symposium (USENIX)*, 2021.

Appendix

Our Modifications in AFL++ for Fuzzing Web Servers. We modified AFL++ to intercept `accept()` and redirect the returned file descriptor to stdin. We terminate the web server process when it attempts to respond to this file descriptor via `send()`. This transforms a stateful web server into a program that processes exactly one network request and terminates, which is an ideal fit for fuzzing with AFL. We also hooked other common networking-related syscalls to ensure that the web server cannot detect the absence of an actual network. Lastly, our harness automatically reasons about the address that `accept()` returns to and uses it as the forking address to accelerate fuzzing. Our harness is a 432-line patch file that may apply to other versions of QEMU.

The rigidity of FirmAFL. We initially planned to adapt FirmAFL's fuzzing engine for our evaluation. However, FirmAFL's fuzzer is tightly coupled with their workflow and thus is difficult to extend to new firmware services. Every FirmAFL target has configuration files that contain key harnessing configuration settings, e.g., maximum execution counts and fork addresses. These settings require manual reverse engineering of the target web servers and cannot be obtained automatically. Moreover, FirmAFL's integration with AFL-QEMU includes hard-coded comparisons against specific target IDs to determine fuzzing behavior.

The rigidity of EQUAFL. EQUAFL has less hardcoded samples-specific logic compared to FirmAFL, but we still found similar manually-inserted hooks to fix rehosted functionality for at least nine of their 66 fuzzable samples. EQUAFL's state synchronization is also tightly coupled with Firmadyne and difficult to adapt to other rehosting solutions. We conclude that both FirmAFL and EQUAFL do not generalize well to new targets, and would require significant manual effort for bootstrapping each new fuzzing target.

Discrepancies in Reported Numbers for FirmAE. Table 10 presents the number of emulated webserver services that successfully achieved each of the four rehosting stages (Unpack, Execute, Connect and Interfact) in our experiment. We evaluated Greenhouse and FirmAE based on the FirmAE paper's dataset, which consists of 1,124 unique firmware images organized over the eight brands (excluding Tenda). The numbers of rehosted services for FirmAE differ from the original numbers reported in the FirmAE paper. This is because we evaluated both platforms under stricter criteria for success based on the stages discussed in Section 5. For example, FirmAE considered a firmware target successfully rehosted if an HTTP request returned without errors or timeouts, which roughly matches the Connect milestone. While this may appear sufficient at first, the check does not filter out cases where a web server may have connectivity but not any actual functionality, as seen among ASUS samples.

Manual crash triaging. Table 11 maps each of the 79 crashes found during our fuzzing of 14 images to 26 legitimate 0-day vulnerabilities. Not all unique crashes found through fuzzing translate to real-world vulnerabilities. Some crashes may be caused by the same root cause. Others may be false positives due to assumptions made in our emulation. For example: to keep things lightweight, our Greenhouse+AFL integration assumes a stateless program target. However, many embedded web services preserve some form of state, such as authenticated sessions. This may lead to crashes when fuzzing that are not reproducible. We mark these crashes as "Not Replicable" in our table. Some crashes also have root causes that are too complex to determine within the time frame of this paper. We mark these crashes as "Untriagable".

		FirmAE				Greenhouse			
Brand	**Initial**	**Unpack**	**Execute**	**Connect**	**Interact**	**Unpack**	**Execute**	**Connect**	**Interact**
ASUS	107	107	107	64	0	107	106	101	101
Belkin	37	37	37	23	5	37	34	25	19
D-Link	263	263	260	241	176	254	236	196	183
Linksys	55	55	53	46	31	53	26	19	18
Netgear	375	375	375	341	266	375	359	265	198
TP-Link	148	148	148	121	98	142	65	16	2
TRENDnet	119	119	112	77	41	97	91	62	47
Zyxel	20	20	20	10	6	20	15	8	6
Total	1,124	1,124	1,112	923	**623**	1,085	932	692	**574**

Table 10: FirmAE- and Greenhouse-rehosted web servers that achieved each of the four rehosting stages (Unpack, Execute, Connect and Interfact) on the FirmAE dataset of 1,124 unique firmware images organized over eight brands (excluding Tenda).

Vuln No.	**Firmware**	**Binary**	**CWE**	**Crashes**
1	AC1450_V1.0.0.6_1.0.3	/usr/sbin/httpd	CWE-822: Untrusted Pointer Dereference	5
2	WN2000RPT_V1.0.1.20	/bin/boa	CWE-822: Untrusted Pointer Dereference	1
-	WNDRMACv2_Firmware_Version_1.0.0.4	/usr/sbin/uhttpd	Untriagable	7
3	DAP_1513_REVA_FIRMWARE_1.01	/bin/webs	CWE-121: Stack-based Buffer Overflow	2
4	DAP_1513_REVA_FIRMWARE_1.01	/bin/webs	CWE-476: NULL Pointer Dereference	2
5	DAP_1513_REVA_FIRMWARE_1.01	/bin/webs	CWE-822: Untrusted Pointer Dereference	1
6	DAP_1513_REVA_FIRMWARE_1.01	/bin/webs	CWE-121: Stack-based Buffer Overflow	1
7	DAP_2330_REVA_FIRMWARE_1.01RC014	/sbin/httpd	CWE-122: Heap-based Buffer Overflow	1
8	DAP_2695_REVA_FIRMWARE_1.11.RC044	/sbin/httpd	CWE-122: Heap-based Buffer Overflow	1
9	DIR_601_REVA_FIRMWARE_1.02	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	1
10	DIR_601_REVA_FIRMWARE_1.02	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	1
11	DIR_825_fw_revb_209EUb09_03_ALL_multi_20130114	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	6
12	DIR_825_fw_revb_209EUb09_03_ALL_multi_20130114	/sbin/httpd	CWE-122: Heap-based Buffer Overflow	2
-	DIR_825_fw_revb_209EUb09_03_ALL_multi_20130114	/sbin/httpd	Not Replicable	1
13	DIR_825_REVB_FIRMWARE_2.03	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	8
14	DIR_825_REVB_FIRMWARE_2.03	/sbin/httpd	CWE-252: Unchecked Return Value	1
15	DIR_825_REVB_FIRMWARE_2.03	/sbin/httpd	CWE-122: Heap-based Buffer Overflow	4
16	FW_RT_AC750_30043808497	/usr/sbin/httpd	CWE-252: Unchecked Return Value	12
17	FW_RT_G32_C1_5002b	/usr/sbin/httpd	CWE-121: Stack-based Buffer Overflow	5
18	FW_RT_G32_C1_5002b	/usr/sbin/httpd	CWE-121: Stack-based Buffer Overflow	3
19	FW_RT_G32_C1_5002b	/usr/sbin/httpd	CWE-121: Stack-based Buffer Overflow	2
20	TEW_632BRPA1_FW1.10B31_	/sbin/httpd	CWE-122: Heap-based Buffer Overflow	3
21	TEW_632BRPA1_FW1.10B31_	/sbin/httpd	CWE-252: Unchecked Return Value	2
22	TEW_632BRPA1_FW1.10B31_	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	3
23	TEW_632BRPA1_FW1.10B31_	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	1
24	TEW_632BRPA1_FW1.10B31_	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	1
25	TEW_632BRPA1_FW1.10B31_	/sbin/httpd	CWE-122: Heap-based Buffer Overflow	1
26	TEW_652BRP_v2.0R_2.00_	/sbin/httpd	CWE-121: Stack-based Buffer Overflow	1

Table 11: Mapping of the 26 reported 0-days to the 79 crashes found through fuzzing 14 firmware images with Greenhouse and AFL++. The number of unique crashes that map to that 0-day are listed with the corresponding firmware, binary and CWE.