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

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* that 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 attempts to rehost our collected 7,111 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 722 N-day vulnerabilities and 24 zero-day vulnerabilities.

1 Introduction

The Internet of Things (IoT) outnumbers humans almost 2-to-1: As of 2022, approximately 14.4 billion connected IoT devices [16] exist out in the wild, and current estimates project the total number to reach 34.2 billion by 2025 [20]. Naturally, the security of these devices is not perfect, with 747 vulnerabilities across 86 different vendors disclosed in the first half of 2022 alone [33]. 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 extremely 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 PCs or 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 RAM), 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 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 either to emulate peripherals [3, 12, 17, 22, 28] or to proxy the communication to peripherals running on real IoT devices [15, 27, 32]. However, the current state of peripheral-aware rehosting techniques only allows for the analysis of firmware based on small, embedded software platforms (such as FreeRTOS or Arduino) or no operating system at all ("bare metal" blobs). Critically, current approaches cannot scale to the analysis of more complex Linux-based firmware, the operating system used by 43% of IoT devices [13].

State-of-the-art rehosting solutions targeting Linux-based firmware currently 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 general device emulation, and booting the firmware sample in a system emulator such as QEMU [4, 20]. However, this implicit concession of rehosting fidelity (e.g., by replacing the embedded kernel) leads to rehosting failures. For example, Firmadyne [4] 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 [20], a refined version of Firmadyne [4], 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, Tenda recently disclosed 10 Buffer Overflow vulnerabilities in network-facing functions of the `httpd` binary of its Linux-based AC21 device, and none of these binaries interact 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 operating systems 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* 2,055 web servers out of 7,111 crawled firmware images and successfully confirmed 597 N-day exploits using the RouterSploit framework [34]. We also integrate our rehosted images with AFL [38] for fuzzing in user-space, achieving a 300% higher throughput during the course of finding the same bugs as the state-of-the-art firmware-fuzzing solution, FirmAFL [39]. We then extended this integration to nine other rehosted targets and found an additional 37 crashes, 24 of which are confirmed 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 device 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 3,326 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 powerful 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 specific and generic *interventions*.
- We conduct a large-scale evaluation on 7,111 unique firmware samples from nine different vendors and *fully rehost* 2,055 web servers. We also demonstrated unique advantages of Greenhouse for vulnerability discovery and assessment by comparing against the state-of-the-art firmware fuzzing solution, FirmAFL.

In the spirit of open science, we will publicly release the source of Greenhouse and research artifacts upon the acceptance of this paper. We are investigating solutions for sharing our firmware data set within the research community without violating device vendors' software distribution restrictions.

2 Background and Motivation

Firmware contains one or more services, where each service may have one or multiple executables that implement the service. *Rehosting* is the process of recreating the behaviors of one or several firmware services on a hardware device (e.g., a router) inside an emulated environment. In this section, we first present state-of-the-art techniques for firmware rehosting (Section 2.1). Next we define the fidelity of rehosting (Section 2.2), and then map rehosting techniques into varied rehosting requirements and identify a research gap (Section 2.3). Finally, we present the motivation for our solution that fills the gap (Section 2.4).

2.1 Rehosting Goals and Techniques

IoT systems can be categorized into three types [11, 24]: **Type-I** devices, which run a general-purpose operating system (OS) and the device is adapted to embedded environments. **Type-II** devices, which have custom OS designed for embedded environments, but still share a distinction between the application layer and the kernel. **Type-III** devices, also known as "monolithic firmware," where the code is a single blob running on the device, and it usually has tight coupling between the firmware and its hardware.

The rigidity of firmware and the hardware it runs on severely limits the types of analysis that researchers can conduct. A key advantage of rehosting is applying diverse analysis techniques to a firmware service, ideally in a scalable and automated manner [11].

Through rehosting, researchers have attempted hardware replacement, behavioral analysis, or engineering compatibility with legacy components [36]. Given the dire situation of IoT-world security, most rehosting studies focus on security analysis, such as automated vulnerability discovery [12, 22, 27, 31, 39] and vulnerability risk assessment (i.e., assessing the real-world impact of an exploit) [4, 20, 40]. These solutions either are sole static analyses [10, 15, 17], or combine static analysis with full-system emulation (for Type-I devices) or full-firmware emulation (for Type-II and III devices) using OS-level emulators (e.g., QEMU-system) [4, 5, 12, 15, 18, 20, 27, 28, 31, 32, 39]. The use of full-system emulators attempts to minimize the discrepancies between a rehosted environment and the original environment (i.e., a real device).

2.2 Rehosting Fidelity

Divergences between the emulated environment and the original can cause a rehosted firmware service to behave differently or even fail to run entirely. Abstractly, we define *fidelity* of a firmware component in an emulated environment as the degree to which it resembles the same component on a real device. We further define the fidelity of static components (e.g., files) as *Extraction Fidelity* and the fidelity of dynamic components (e.g., the runtime behavior of a firmware service) as *Execution Fidelity*.

Extraction Fidelity. Static components in a firmware image include files (or raw data when no file systems are used), as well as data that is stored on the image or in hardware (e.g., NVRAM). Researchers usually obtain these components by extracting data from downloaded firmware images or physical devices. These can be supplemented with meta-information, such as debug symbols and source code of open-source libraries, and common manufacturer-specific information (e.g., memory layout). 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.

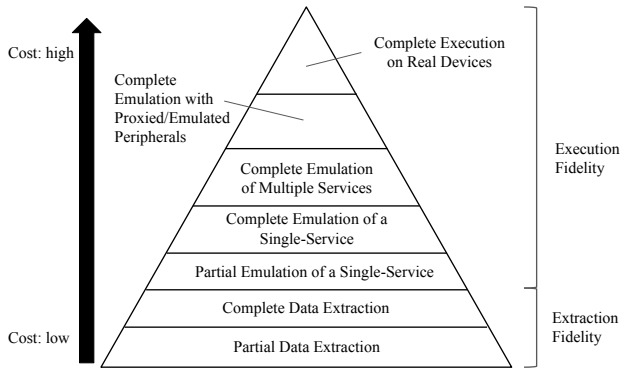

Figure 1: Hierarchy of rehosting requirements. Both rehosting fidelity and costs increase from bottom to top.

Execution Fidelity. On most embedded devices, 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, reading sensor data, or sending signals. Examples of low Execution Fidelity in an emulated service include unintended behaviors, missing pieces or broken protocols during communication that hamper access to the service (e.g., failing to log in to an emulated web portal), or even prematurely exiting or crashing the service itself. Achieving a high level of Execution Fidelity is critical during vulnerability discovery and assessment, as behaviors caused by low Execution Fidelity may lead to false positive alarms (if any crashes are treated as vulnerability indicators) or false negative alarms (when vulnerable code does not run at all in emulation).

Execution Fidelity and Extraction Fidelity. High Extraction Fidelity is a stepping stone toward 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 bottom levels are more fundamental and easier to achieve, and the upper levels are usually much harder to achieve.

2.3 Rehosting and Execution Fidelity

Table 1 shows the rehosting requirement of existing rehosting techniques. Existing solutions either skew towards full-system emulation to perform dynamic analysis, or sidestep the execution fidelity issue entirely to focus on pure static analysis. Although full-system emulation enables high Execution Fidelity, it often comes with high overhead and a high cost, especially for dynamic analysis techniques such as fuzzing.

Researchers have proposed workarounds to reduce the overhead involved in full-system rehosting, but in practice this usually results in a tight coupling between the fuzzer and the rehosted firmware targets, which hampers the generality the

Requirement	Extraction Fidelity	Execution Fidelity	OS Emulation	Peripheral Emulation	Require Devices	Runtime Overhead	Techniques
Complete Emulation w/ Peripherals	High	High	✓	✓	✓	High	Pretender [15], Charm [32], Frankenstein [27], HALucinator [5], P2IM [12], Jetset [18], FirmFuzz [31], Fuzzware [28], μEmu [40]
Complete Emulation	High	High	✓	✗	✗	High	FirmAE [20], Firmadyne [4], FirmAFL [39]
Partial Emulation, Single Service	High	High	✗	✗	✗	Low	Greenhouse
Data Extraction	Low	-	✗	✗	✗	None	FirmUSB [17], Karonte [10]

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

of rehosting techniques. For example, during the evaluation of Greenhouse, we found FirmAFL [39] requires intensive manual effort for harnessing *each* firmware target (details in Section 7). Additionally, configuration, software, and hardware discrepancies between the real device and an emulation environment are oftentimes inevitable. Increasing the execution fidelity by resolving these discrepancies piecemeal requires significant manual effort for each device.

2.4 Motivation

Most vulnerabilities in firmware are unrelated to peripherals. We sampled 100 firmware CVEs reported on NVD [25] within the last two years and found that only 14% of them were related to hardware. Many of the remaining ones are intrinsically independent of hard-to-emulate, device-specific characteristics and features. For example, Tenda recently disclosed 10 buffer overflow vulnerabilities in network-facing functions of the `httpd` executable of its AC21 device, none of which interact with peripherals. However, these executables may encounter *roadblocks*, such as loading configuration entries, communicating with a peripheral, or invoking device-specific features, *before* reaching the vulnerable program points. For the purpose of vulnerability discovery and risk assessment, our insight is that we can automatically eliminate many of these roadblocks by directly manipulating the binary code.

Over the past few years, researchers recognize that many firmware vulnerabilities do not require a faithful emulation of peripherals. Partial emulations using stubs [4, 20, 31] or models [12, 15, 40] have been used to find vulnerabilities in firmware through dynamic analysis. We generalize this idea by applying it onto all *roadblocks* in emulated firmware services. Instead of tricking a service into communicating with a stub, we simply patch the binary code to prevent services from tripping on these roadblocks.

Further, by observing that 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 achieve sufficiently high Execution Fidelity *without* the expensive full-system emulation. This allows us to significantly improve the execution speed and the portability of rehosted services, and enables us to integrate with existing tools such as AFL [38] and RouterSploit [34].

3 Single-Service Rehosting

Greenhouse focuses on Type-I IoT devices (as defined in Section 2.1 that use a Linux-based operating system (OS). We further select routers because they represent one of the largest and most commonly studied subset of IoT devices in the wild. We limit ourselves to firmware images that run on the following 32-bit architectures: MIPS, MIPSEL, ARM, ARMEB, and X86, as they represent the majority of publicly available firmware images in the wild.

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 binary may invoke several scripts to dynamically generate HTML content for its users. The web server is the primary process, which, together with the scripts, constitutes a single service.

Single-service rehosting focuses on rehosting these type of firmware services. We decide to target web servers in this paper as they are the most commonly accessible service from LAN ports of routers across multiple vendors, and they are the gateway for many CGI binaries. To minimize the execution overhead, we rehost services in user mode (i.e., without full-system emulation): We use QEMU-user to emulate service binaries inside a `chroot` file system, which we term *single-user rehosting*, as opposed to full-system rehosting (via QEMU-system) that other solutions use. Single-user 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-user rehosting of single firmware services. It comprises three main components: the Runner, Checker, and Fixer. Supplement-

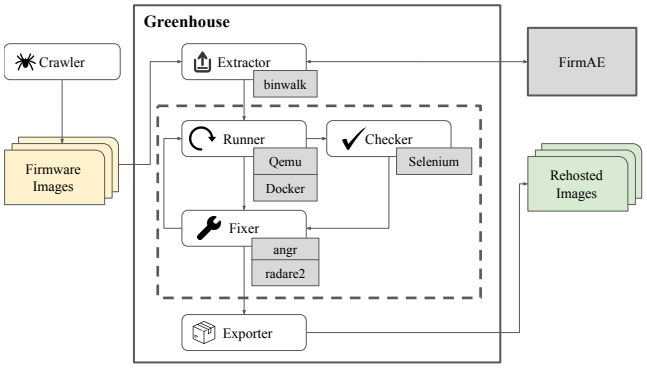

Figure 2: An overview of the Greenhouse pipeline that shows the flow of data between components as a firmware image progresses through the iterative rehosting pipeline, along with key libraries that each component relies on.

ing these main components are an Extractor component that 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 builds the 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.

4.1 File System Extraction

Extraction is done via Binwalk with all optional extraction tools installed. We call Binwalk with both `-M` (Mashotrya) and `--preserve-symlinks` to recursively extract the root file system from the firmware image and preserve symlinks. The Extractor then performs static analysis to identify the web-server binary, its architecture, and all C standard libraries against which it was linked. Using this information, it applies a number of common *interventions* to the file system (discussed in Section 6) that generally improve the Extraction Fidelity of the extracted file system. This stage completes once the architecture, libraries, and the target web server are identified and passed to the Runner.

Although Greenhouse focuses on identifying and rehosting web servers, it supports rehosting *any* type of single service.

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 TCP ports to test. It tests basic connectivity to the web server by performing an HTTP GET request to each network address. For every request that receives a response, the Checker verifies that the HTTP status code is either 200 (OK) or 401 (Unauthorized). We then use a heuristic-based Selenium script to detect the existence of login portals, and we attempt to log in if one exists.

Regardless of the login result, the same Selenium script then loads the landing page of the web server. We check for common error messages, empty HTML pages, and other indicators of misconfiguration that are generalizable to the broad set of router brands that we analyze. The Checker uses these indicators to determine the level of Execution Fidelity of the rehosted service (see Section 5).

The Checker reports the results of its probing, e.g., the resulting status codes, timeouts, and any HTTP headers used in the final GET request, to the Fixer. While the Checker only tests web-server-specific behaviors, Greenhouse is generic enough that users may plug-in other Checkers and Fixers 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 as well as probing results from the Checker to diagnose potential roadblocks that limit Execution Fidelity. For each roadblock, the Fixer applies the corresponding intervention, which we will detail in Section 6. For interventions

that requires binary manipulation, Greenhouse uses radare2 to assemble generic patches for different architectures [2].

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, 30 in our experiments). The rehosted file system is then packaged by the Exporter.

4.5 Containerization

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 Milestones

Each rehosting loop (Running, Checking, and Fixing) attempts to increase the level of Execution Fidelity for the target service. Conceptually, Greenhouse breaks down the Execution Fidelity that the target service achieves into four *milestones*: unpack, execute, connect, and interact.

Separating each of these milestones are *rehosting roadblocks* that hinder progression to the next level of Execution Fidelity. These roadblocks are discrepancies between the emulated environment that our rehosted service runs in and the physical device. We empirically determine a set of common roadblocks through manually examining hundreds of firmware samples and develop a set of *interventions* for as many of them as possible. The types of roadblocks encountered, and the interventions deployed by Greenhouse to resolve them, are discussed in greater detail in Section 6.

By iteratively applying these interventions to the web server and the surrounding file system environment, Greenhouse can drive the Extraction Fidelity and Execution Fidelity up to the level where dynamic analysis techniques (e.g., fuzzing) can be effectively applied.

Milestone 1: Unpack. At this stage, Greenhouse does not differ much from other full-system rehosting techniques. To begin rehosting, we 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.

Milestone 2: Execute. Rather than trying to emulate the entire boot environment, Greenhouse locates an executable binary that is associated with the router web server using a whitelist of common web server names. Greenhouse verifies if the identified web server 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 milestone achieved even if the web server immediately exits or crashes, as long as it executes.

Milestone 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 exploits for firmware involve communicating with its network-facing services.

We consider this milestone achieved if we can receive an HTTP response after sending an HTTP request to the rehosted service without it terminating, timing-out, or prematurely crashing. 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 HTTP request parsing code).

Milestone 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 the web server.

To determine if the web server is running at a high Execution Fidelity level, Greenhouse performs some basic interactions with the web service. It checks status codes of HTTP responses and compares the returned content against a set of pre-set error strings to identify malfunctioning backends. We use Selenium to load dynamic content and attempt some common login protocols. While these checks are not a complete test of the firmware web server's behavior, our evaluation in Section 7 will show that rehosting a firmware service to this milestone 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

[1]Prior work referred to interventions as *arbitrations* [20], *augmenting* [39], *mitigations* [26] and *refinement* [11]

automatable interventions for them, which we implement in Greenhouse.

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.

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.

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-binary rehosting, as full-system rehosting abstracts the issue to the initialization scripts that the kernel runs at boot.

Peripheral Access (R3). A common roadblock that many emulations, particularly ones focused on targeting the peripherals themselves, try to address. Firmware web servers may try to communicate with hardware peripherals in a variety of ways ranging from access an expected `/dev` file to directly accessing a reserved region of memory. In our limited emulation these methods 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. NVRAM usually contains default configuration data that is necessary for the firmware image to start up and function.

Hard-coded Network Devices (R5). Web servers usually have a hardcoded "default" IP addresses or device name that they bind to by default. If a network device with that address or name is not present, the web server fails. Included in this category are web servers that communicate primarily over IPv6, as it is not supported in our emulation.

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.

Non-Critical 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. These checks share the common behavior of exiting if conditions in the check are not met.

Environment Mangling (R8). Many firmware binaries are 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 our QEMU emulator that gathers critical data to identify potential roadblocks and interventions needed.

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.

Thus, Greenhouse implements a "best-effort" intervention system that tries to fulfill as many roadblocks as possible before falling back to a general *patching system* that bypasses the roadblock instead. What follows is the list of best-effort interventions that Greenhouse performs and their corresponding roadblocks:

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 expected paths and create a corresponding empty file/folder in the rehosted file-system. If we are instead able to find a backup of the file, or if the file path was misplaced in our runtime environment, we copy it to the new location.

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.

File-system Snapshot (I3) - *[R1, R2, R7].* We leverage FirmAE to run a full-system emulation of the image and obtain a "file-system snapshot" of the runtime processes, files, and QEMU serial logs. From this snapshot we can extract any files that are generated at boot, command-line arguments passed to the target binary, and any configuration data loaded into NVRAM or other supporting background processes.

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 [8]. We extend it to implement *nvram-set* functionality and provide wrappers for a wider range of known NVRAM functions. We cross-compile a standalone version of the library and replace the default `libnvram.so`. 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 our emulation based on

the NVRAM values from the file-system snapshot or from a list of common defaults.

Runtime ArgParser (I5) - *[R2].* We provide a fallback intervention if command-line arguments for the firmware binary cannot be obtained from a file-system snapshot. 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.

Dummy Network Devices (I6) - *[R5].* A modified version of the QEMU `bind()` syscall logs whenever a bind is attempted on a particular network address or device name. Greenhouse then configures the subsequent Docker containers in our emulation with the corresponding network bridge. In case the binary is intended to be run outside a Docker container Greenhouse also provides an automatically generated script that uses the Linux `ip` binary to create the necessary dummy devices.

Background Script Plugins (I7) - *[R6].* Although a general solution to handle *multi-service* targets is considered out of scope, Greenhouse provides a specific solution for emulations that communicate with other processes running in the background as a core part of their functionality. Greenhouse provides a framework for plugin scripts that specify a heuristic for identifying the binary and a means of running it. These scripts can also specify basic shell commands that can be run inside FirmAE's emulation to retrieve runtime configuration data. As a demonstration of this, we provide plugins for two such extra services, `xmldb` and `config`, which are used by D-Link and Netgear web servers 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 on which Greenhouse is running, such as the Kubernetes cluster that we used to conduct our large-scale rehosting and analysis. Hence, we implemented a workaround by modifying the `bind()` syscall inside QEMU to convert binds to IPv6 addresses to the IPv4 address 0.0.0.0. We also add a simple array to de-conflict binds to the same port on 0.0.0.0, to handle the case where multiple addresses are converted and bound to 0.0.0.0.

Patching `sysinfo()` (I9) - *[R7].* Some routers adaptively disable services based on currently load, which makes it difficult to perform dynamic analysis. 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 for the current CPU load, thus preventing these behaviors.

Logging Behavior (I10) - *[R8].* Our emulation in user-mode shares file descriptors with the emulated service. 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. A context-sensitive CFG is constructed using a static CFG, where the calling context under which a particular block is called is taken into account. Thus, each node in the graph is differentiated not just by its address, but also the addresses of all the blocks that call it in the static CFG. This allows us to easily map the execution trace of a binary to the CFG for the purposes of our 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 Non-Critical 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 Non-Critical 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 7.3)
- What factors impact the rehosting performance of Greenhouse? (Section 7.4)
- Does Greenhouse reach a level of Execution Fidelity that enables vulnerability discovery and risk assessment? (Section 7.5)
- How much does Greenhouse-rehosted service improve the fuzzing performance? (Section 7.6)

7.1 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 8GB of RAM. We modified FirmAE to run on a Kubernetes cluster. Our modifications will be released when we open source all research artifacts.

7.2 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. Then, we merged the remaining images with FirmAE's data set and removed duplicates. Our final collection of firmware images consists of 7,111 unique firmware images. This collection is 6.3x FirmAE's data set (containing 1,124 images), and 1.3x of the number of router images in Firmadyne's data set (which contains 5,507 images collected in 2015 from the nine vendors)[2].

7.3 Firmware Rehosting Results

We compare Greenhouse against its closest predecessor, FirmAE. Because FirmAE is based on Firmadyne, and FirmAE already evaluates against Firmadyne in their paper, we do not include Firmadyne in this experiment.

Determining levels of Execution Fidelity. We use the Checker component (described in Section 4.3) to determine the Execution Fidelity in terms of the milestones described in Section 5. In addition, we also parse the logs of both Greenhouse and FirmAE to determine the degree to which Unpack and Execute succeeded for the image. For Greenhouse, we consider Unpack successful if we find a web server, and a successful Execute if we can run it in QEMU-user. For FirmAE, we consider Unpack successful if it can find and mount a file system image, and successful Execute if it can boot the image in QEMU-system.

Results. Table 2 shows the numbers of rehosted firmware services (or images for FirmAE) that reached each rehosting milestone under Greenhouse and FirmAE. Overall, the number of Greenhouse-rehosted firmware services is comparable to that of FirmAE (2,055 vs. 2,023). 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).

Overlaps between rehosted services. To understand the results, we examined the overlap between FirmAE- and Greenhouse-rehosted services. Interestingly, as shown in Table 3, the set of firmware services that Greenhouse fully rehosted has little overlap with the ones that FirmAE successfully rehosted. Together, FirmAE and Greenhouse are able to rehost 3,326 out of 7,111 firmware services, which cover nearly 50% more than either solution can individually rehost. This shows that Greenhouse is not a subset of full-system emulation techniques, but rather a technique that handles unique rehosting obstacles that full-system emulation techniques cannot.

[2]While Firmadyne's data set has more firmware images (23,035), the majority of them are not router firmware images.

		FirmAE				Greenhouse			
Brand	**Initial**	**Unpack**	**Execute**	**Connect**	**Interact**	**Unpack**	**Execute**	**Connect**	**Interact**
ASUS	830	828	813	453	5	828	814	721	684
Belkin	63	63	56	19	9	62	62	38	32
D-Link	1,467	1,436	1,077	562	435	1,090	1,067	488	423
Linksys	84	84	78	35	24	81	78	28	26
Netgear	2,796	2,669	2,487	1,289	970	2,300	2,224	1,060	567
Tenda	174	163	143	28	12	164	163	75	61
TP-Link	945	937	863	466	372	842	823	157	91
TRENDnet	732	717	664	275	193	637	629	280	164
Zyxel	20	20	20	5	3	20	19	12	7
Total	7,111	6,917	6,201	3,132	**2,023**	6,024	5,879	2,859	**2,055**

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

	Successfully Rehosted By		
	FirmAE only	**Both**	**Greenhouse only**
ASUS	4	1	680
Belkin	7	2	25
D-Link	330	105	93
Linksys	11	13	15
Netgear	270	700	297
Tenda	10	2	51
TP-Link	25	347	66
TRENDnet	94	99	70
Zyxel	1	2	6
Total	1,271	752	1,303

Table 3: 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.

7.4 Case Studies

7.4.1 ASUS Firmware

We examined ASUS firmware services for which Greenhouse rehosted 681 out of 830 to the Interact milestone while FirmAE only rehosted 5. While both Greenhouse and FirmAE rehosted similar numbers of services to the Execute milestone, FirmAE was unable to Connect half of them, and only 5 were able to Interact.

Configuration Reuse. We first analyzed 300 services that FirmAE rehosted to Execute but was unable to Connect but Greenhouse successfully rehosted until 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 219 of them, most notably NVRAM data from other Netgear images in our collection. Critical NVRAM values include `lan_ipaddr` and `env_path` that directly affect the web server's execution. By reusing configuration information from other images in our collection, Greenhouse rehosted more services to the Connect milestone.

Power of the Checker. We then analyzed 383 services that FirmAE rehosted until Connect could not Interact, and that Greenhouse rehosted to Interact. This represents the impact of Greenhouse's interventions on the Execution Fidelity of rehosted services. For nearly all services (310/383), the emulated service in FirmAE returned an HTTP status code of 200, but the actual web pages displayed file-not-found error messages[3]. Greenhouse also encountered these errors during rehosting, however Greenhouse successfully detected that the rehosting was insufficient due to the stricter, Selenium-based checking criteria employed by the Checker, and performed more iterations of interventions on these services. In the next few iterations, Greenhouse fixed all encountered roadblocks, including bypassing missing files (e.g., `QIS_wizard.html` or `cert.pem`).

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.

7.4.2 Tenda Firmware

Greenhouse used the Wait Loop Patcher to patch 26 Tenda services and applied the Premature Exit Patcher on all 51 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 [35]. When unable to access CFM, web servers exit with the error message "connect cfm failed!" before initiating any network behaviors. By patching the check leading to exit, Greenhouse forced the execution into network-facing code.

7.4.3 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.

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

7.5 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. We selected all rehosted services that reached at least Connect for both Greenhouse and FirmAE and replayed 125 known N-day exploits against each rehosted service.

Result. As seen in Table 4, RouterSploit exploited 722 known vulnerabilities across 2,859 Greenhouse-rehosted firmware services. Meanwhile, RouterSploit found 617 known exploits on 3,132 samples rehosted by FirmAE. Despite not conducting full-system emulation or modeling peripherals, Greenhouse-rehosted services are sufficient to use for vulnerability risk assessment.

7.6 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 fuzz a selected subset with AFL to find vulnerabilities. We randomly selected seven Greenhouse-rehosted services that reached the Interact milestone, along with three firmware images with known web server vulnerabilities in the FirmAFL data set that Greenhouse successfully rehosted, for a total of ten fuzzing targets. Additionally, to evaluate performance improvements of user-space fuzzing over fuzzing with augmented process emulation (as discussed in FirmAFL [39]), we fuzzed three rehosted FirmAFL services using both Greenhouse+AFL and FirmAFL.

7.6.1 Simple Fork-server Fuzzer

We chose AFL as the fuzzer and built a generic fuzzing harness for web servers, which emulates client connections in AFL-QEMU. 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 easily applies to other QEMU implementations.

The rigidity of FirmAFL. We initially planned to adapt FirmAFL's fuzzing engine for this experiment. 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. We concluded that FirmAFL's approach does not generalize to new targets, and would require significant manual effort for bootstrapping each new fuzzing target.

7.6.2 Performance Evaluation

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, 500GB SSD, and 270GB of RAM. For the three FirmAFL targets, we ran our fuzzer and FirmAFL's fuzzer ten times each and measured both the execution speed and time to find the same crashes that were reported by FirmAFL. We then fuzzed all ten targets for 24 hours and measured the numbers of additional unique crashes that were found for each target.

Table 5 shows the performance of fuzzing using AFL on Greenhouse single-user rehosting against FirmAFL for the three rehosted `httpd` servers. In all cases, our fuzzer could find the same vulnerability reported by FirmAFL on Greenhouse-rehosted targets. On average, fuzzing in user space achieves a 3x improvement in speed compared to augmented process emulation[4].

We filter crashing inputs using `tmin` and `md5sum` to identify unique crashes that are worth looking into. We found 37 unique crashes across ten services, which we manually verified. We also ensured that the vulnerabilities were not introduced by binary patching in Greenhouse. We confirmed that 24 out of 37 are legitimate 0-day vulnerabilities in firmware services, and are in the process of responsible disclosure.

7.6.3 Replaying Exploits

FirmAFL reported 15 CVEs over nine unique firmware images, three of which we discovered via fuzzing faster than FirmAFL. Of the 12 remaining CVEs reported by FirmAFL, Greenhouse successfully rehosted three to the Interact milestone, seven to the Connect milestone, and one to the Execute milestone. The others are local services that do not depend on web servers. Because our evaluation is about the rehosting performance of Greenhouse, writing additional fuzzing harnesses for local services is out of scope.

However, to demonstrate that rehosting and fuzzing these binaries is possible with user-mode single-process rehosting, we manually replayed all exploits discovered by FirmAFL against Greenhouse-rehosted targets. We successfully replayed exploits against all firmware services that were rehosted to the Connect milestone, and manually verified that they all triggered intended crashes. In total, we replayed 13 out of 15 CVEs reported by FirmAFL, showing that the rehosting of Greenhouse achieves a sufficiently high level of

[4]The sample TEW-632BRP1.010B32 timed out after fuzzing with FirmAFL for 24 hours. We did not receive assistance from the authors regarding this sample before the submission deadline, and we will continue to attempt to contact the authors.

	FirmAE				Greenhouse			
	Path Traversal	**Command Injection**	**Info Leak**	**Auth Bypass**	**Path Traversal**	**Command Injection**	**Info Leak**	**Auth Bypass**
ASUS	0	12	0	0	0	0	0	0
Belkin	0	2	3	3	13	4	11	11
D-Link	0	230	237	18	3	161	273	12
Linksys	0	1	0	0	0	1	0	0
Netgear	0	69	3	0	153	9	33	0
Tenda	0	0	0	0	0	0	0	0
TP-Link	0	0	0	0	0	0	0	0
TRENDnet	0	9	30	0	12	8	18	0
Zyxel	0	0	0	0	0	0	0	0
Total	**0**	**323**	**273**	**21**	**181**	**183**	**335**	**23**

Table 4: Breakdown of running 125 N-day exploits on stage 3 rehosted images using RouterSploit. Each cell shows the number of N-day exploits of a given type that successfully exploit a rehosted image for a given brand under FirmAE and Greenhouse respectively.

	Greenhouse		FirmAFL	
ID	**time to crash (s)**	**execution speed (#/s)**	**time to crash (s)**	**execution speed (#/s)**
DAP-2695	7,400.12	772.85	11,354.90	242.82
DIR-825	1,520.58	548.84	30,082.00	165.13
TEW-632BRP	1,245.01	177.12	*	*

Table 5: Fuzzing Performance of Greenhouse+AFL versus FirmAFL. We report executions per second and the total time taken to reach the same crashes reported by FirmAFL.

	Tmin inputs (total crashes)	**Unique crashes (md5 + manual)**	**Unique vulns**
DAP-26951.11.RC044	260	1	1
DIR-8252.02	410	5	4
TEW-632BRP1.01	323	6	4
AC1450_V1.0.0.6	19	1	1
DAP_1513_REVA_1.01	32	1	1
DIR-601_REVA_1.02	41	7	4
DIR-825_REVB_2.03	52	6	2
FWRT_AC51U	0	2	2
FWRT_G32_C1	66	5	2
TEW_652BRPv2.0R	22	3	3
Total	**1,225**	**37**	**24**

Table 6: Vulnerabilities found through fuzzing ten firmware images rehosted with Greenhouse+AFL.

Execution Fidelity for vulnerability discovery and risk assessment.

8 Discussion

8.1 Discrepancies in Reported Numbers

Because our data set includes the original set of firmware images used by FirmAE, we provide a breakdown of the rehosting result for the 1,124 FirmAE targets (See Table 7). 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 milestones 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.

8.2 Limitations

Encrypted firmware. Greenhouse needs a high Extraction Fidelity to perform its iterative rehosting approach. Our patching approaching 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. Specifically, user-mode QEMU 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 7,111 cases, about 2% of our 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. In particular, a number of MIPS targets (68, or 0.9% of our collection) crashed due to assertion failures in angr when creating context-sensitive 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 duringg rehosting, which we plan to address as an engineering problem in the future.

9 Related Work

Large-Scale Emulation. Previous work in large-scale emulation of firmware images almost entirely focuses on full-system emulation. Costin et al. [7] studied 1,925 firmware images over a variety of COTS embedded devices using full-system emulation with common open-source web penetration tools to discover vulnerabilities in the embedded web server. Platforms such as Firmadyne [4], Avatar [37], and Avatar2 [23] automated this process using OS emulators such as QEMU to rehost large sets of firmware images. FirmAE [20] directly built upon this work and formalized the approach in which an extracted firmware image is modified to suit its emulation environment, via *arbitration techniques*.

While these works demonstrated how full-system emulation could be used to achieve large-scale analysis of firmware, they also showed how the fidelity of the emulated environment limits analysis of the target. Other publications have thus focused on improving the fidelity of full-system emulation to include missing components such as peripheral support. Approaches such as PROSPECT [19], CHARM [32], and SURROGATES [21] focus on optimizations for hardware-in-the-loop implementations. Others like Laelaps [3] and Jetset [18] go the other way, using symbolic execution to extrapolate peripheral behavior for their models. Meanwhile, works like HALucinator [5] and DICE [22] emulate Hardware Abstraction Layers (HALs) and the DMA (Direct Memory Access) channel to create a more generalized approach. Nearly all of them use dynamic analysis techniques like fuzzing to evaluate the effectiveness of their implementations.

Greenhouse is more precise in what parts of the firmware we emulate. Instead of being forced to adapt the emulation to the firmware, Greenhouse tailors the firmware image to the environment. Unlike FirmAE, Greenhouse is willing to iteratively add as many interventions as needed, including to patching the binary itself.

This is similar to the approach taken by ARI [26] that builds upon Firmadyne and applies their own set of "interventions" designed to handle different failure cases over 1,709 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. Unlike Greenhouse, ARI still uses Firmadyne's full-system emulation approach to rehost firmware images. Thus, ARI and Greenhouse can be considered orthogonal works, with ARI applying iterative interventions to overcome rehosting challenges in full-system emulations, while Greenhouse explores the challenges of single-service rehosting in user-space.

Another similar work is FirmAFL [39], which hybridizes user-space execution with full-system emulation to gain an 8.2 times improvement in fuzzing performance. The platform achieves this by switching between user and system modes during emulation, but as discussed in our Evaluation (Section 7), requires significant manual effort to integrate with its dataset. Greenhouse emulates a firmware service entirely in user-space, while being generalizable to a much larger set of firmware images and analysis tools.

Other Analysis Approaches. While fuzzing is the most common approach for finding vulnerabilities, other works have achieved notable success via other analysis techniques. Costin et al. [6] performed static analysis on 32,000 firmware images by combining a correlation engine with simple keyword searches. This approach proved surprisingly effective at finding multiple vulnerabilities, including simple backdoors. Recent papers analyze programs by symbolically executing them to look for control flow bugs [9, 17, 29] or statically analyzing the interactions between multiple binaries [10] to detect insecure data flows. While Greenhouse is primarily targeted at rehosting for dynamic analysis, it does make use of static analysis to target its interventions.

Monolithic. Though Greenhouse is limited to Type-I, Linux-based firmware images, recent work have made significant strides in automating the analysis of monolithic firmware. Heapster [14] identifies the heap library used by the image for memory allocation and uses symbolic execution to detect potential vulnerabilities. PRETENDER [15] models MMIO behavior by tracing behavior on the actual device, while P2IM [12] tries to exhaustively probe for MMIO to generate a model. μEmu [40] 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. 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 7,111 Type-I firmware images and rehost 2,055 of them to the minimal level of usability under our new criteria. Using existing analysis tools like RouterSploit and AFL, we find 722 N-day exploits and 24 0-day vulnerabilities. This demonstrates both the feasibility of single-service, user-space emulation in creating usable emulated images for dynamic analysis.

References

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

[2] Sergi Alvarez. Radare2. https://rada.re/n/. (Accessed on 2022-11-08).

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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).

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2014.

[20] 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.

[21] Karl Koscher, Tadayoshi Kohno, and David Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[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 *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, 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. 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] 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.

[32] 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.

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

[34] threat9. routersploit. https://github.com/threat9/routersploit. (Accessed on 2022-10-08).

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

[36] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. volume 54, pages 1–36. ACM New York, NY, USA, 2021.

[37] 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.

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

[39] 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.

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

Appendix

Webserver Service Rehosting Table 7 presents the number of emulated webserver services that successfully achieved each of the 4 rehosting milestones (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 8 brands (excluding Tenda).

		FirmAE				Greenhouse			
Brand	**Initial**	**Unpack**	**Execute**	**Connect**	**Interact**	**Unpack**	**Execute**	**Connect**	**Interact**
ASUS	107	107	105	40	0	107	106	104	103
Belkin	37	37	36	12	6	37	37	19	18
D-Link	262	262	258	196	143	258	258	176	159
Linksys	55	55	52	29	21	53	53	22	21
Netgear	375	375	373	268	221	372	366	219	123
TP-Link	148	148	145	106	88	142	140	23	6
TRENDnet	119	119	112	51	34	113	112	46	30
Zyxel	20	20	20	5	3	20	19	12	7
Total	1,123	1,123	1,101	707	**516**	1,102	1,091	621	**467**

Table 7: Numbers of FirmAE- and Greenhouse-rehosted web servers that achieved each of the four rehosting milestones (Unpack, Execute, Connect and Interfact) on the FirmAE dataset of 1,124 unique firmware images organized over eight brands (excluding Tenda).