User-Space Dependency-Aware Rehosting for Linux-Based Firmware Binaries

Chuan Qin[1,2,3*†],Cen Zhang[3*],Yaowen Zheng[1],Puzhuo Liu[4,5],Jian Zhang[3],Yeting Li[1],
Weidong Zhang[1‡],Yang Liu[3],Limin Sun[1,2‡]
[1]Institute of Information Engineering, CAS, China [2]School of Cyber Security, UCAS, China
{qinchuan, zhengyaowen, liyeting, zhangweidong, sunlimin}@iie.ac.cn
[3]Nanyang Technological University, Singapore [4]Tsinghua University, China [5]Ant Group, China
cen001@e.ntu.edu.sg, {jian_zhang, yangliu}@ntu.edu.sg liupuzhuo.lpz@antgroup.com

***Abstract*—Firmware rehosting is a fundamental emulation technique that enables dynamic analysis of firmware binaries at scale. Successfully rehosting Linux-based firmware services requires proper emulation of both system-level functionalities like device interfaces and user-space dependencies such as configuration files, inter-process communications. However, existing solutions inadequately leverage user-space knowledge. The *init routine*, which is the first user-space process sets up operating environments, is often incompletely executed, leading to incomplete initialization. Besides, all emulation failures are treated uniformly, failing to distinguish between direct system-level emulation issues and their indirect effects on user-space dependencies.**

To fill this gap, we developed FIRMWELL, a framework which first models firmware rehosting as the coordinated emulation of both the target binary and its user-space dependencies. It first rehosts the *init routine* for environment construction and then launches the target, which is a procedure that typically involves more than one hundred processes. When emulation failures occur, FIRMWELL identifies the blocking process, analyzes incorrectly emulated resources, and applies targeted fixes. The key strategy is to address user-space dependency failures by correcting the underlying system-level emulation errors, while employing program analysis for precise resource value inference. In evaluation of 14,049 firmware images, FIRMWELL successfully rehosted 6,490 services, outperforming state-of-the-art by 1.6 - 8x (3,581 for FIRMAE, 3,962 for GREENHOUSE, and 810 for PANDAWAN), while reducing average rehosting time by 1.8 - 8.4x (12 vs. 22, 74, and 101 minutes). FIRMWELL was applied to fuzz 1,043 firmware images, uncovering 67 zero-day vulnerabilities with ten assigned CVE identifiers.

I. INTRODUCTION

The rapid increase in the number of Internet of Things (IoT) devices and the potential vulnerabilities in their firmware pose significant security risks [1], [2]. Dynamic analysis techniques like fuzzing are particularly effective for discovering these vulnerabilities due to their high detection rates and zero false-positive results. However, performing dynamic analysis physically on every IoT device is impractical due to the limited scalability. To overcome these challenges, a more feasible approach is firmware rehosting [3], [4]. This technique lets the IoT firmware be effectively emulated on high-performance servers, making dynamic analysis available and highly scalable without the constraints associated with physical devices.

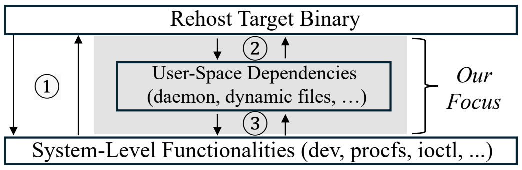

Rehost Target Binary
①
②
③
User-Space Dependencies (daemon, dynamic files, ...)
Our Focus
System-Level Functionalities (dev, procfs, ioctl, ...)

Fig. 1: Overview of Target Binary Rehosting.

Rehosting a Linux-based firmware service requires creating an emulated environment that supports all resources accessed by the target binary. As illustrated in Figure 1, these resources fall into two categories: system-level functionalities and user-space dependencies. The former includes kernel services and hardware peripherals accessed through system calls, device drivers (`ioctl`), and special filesystems like `procfs` and `devfs` (Figure 1 - ①). The latter comprises resources created during system initialization, including configuration files, environment variables, and background services that enable IPC (inter-process communication, Figure 1 - ②). These dependencies are established by the *init routine*[1], which prepares the runtime environment for all subsequent programs. Successful firmware rehosting requires proper emulation of both resource types.

Existing firmware rehosting approaches fall into two categories. The first category employs full-system emulation [6], [7], replacing the original firmware kernel with a pre-built, generic kernel that runs in QEMU. These kernels incorporate heuristics to handle common device-specific features like MTD devices and custom `ioctl` operations. After booting, they rely on the *init routine* to automatically set

[*] Both authors contributed equally to this work.
[†] The work was done while visiting Nanyang Technological University.
[‡] Corresponding author: Weidong Zhang, Limin Sun.

[1]*init routine* [5] is the first user-space process executed after kernel boot, responsible for setting up the user-space environment.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230249
www.ndss-symposium.org

up user-space dependencies. However, this approach often fails in practice because generic kernels cannot accommodate the diverse vendor-specific customizations found across different firmware. The second category, proposed by GREENHOUSE [8], takes a different approach by only executing the target binary in user-mode emulation and iteratively fixing rehosting failures. GREENHOUSE empirically summarizes common failures as roadblocks and applies predefined patterns to resolve them, complementing full-system approaches by successfully rehosting a different subset of firmware.

Despite their progress, existing approaches underutilize user-space dependency knowledge for firmware rehosting, leading to two fundamental limitations. **First, they fail to ensure complete execution of the *init routine*.** The *init routine* establishes essential dependencies for all user-space programs, including configuration files, environment variables, and IPC channels. However, existing approaches treat it as either error-free (full-system emulation) or entirely optional (user-mode emulation). In practice, the *init routine* involves hundreds of processes, and emulation errors in any of these processes, even those unrelated to the target service, can halt the entire initialization. Without dedicated support for completing the *init routine*, critical dependencies remain uninitialized, causing rehosting failures even when the target service could otherwise run successfully. **Second, current approaches apply superficial fixes to user-space emulation errors.** When rehosting fails due to user-space issues like missing files or broken IPC connections, these are typically symptoms of underlying system-level emulation errors. For instance, a web server may fail to find its configuration file since the configuration daemon attempted to bind to a non-existent network device within the emulated environment. Existing approaches employ surface-level fixes such as creating dummy files or patching binaries, without identifying or correcting the actual emulation errors. Such superficial fixes fail when the target service depends on the actual functionality these resources provide.

To address these limitations, we present FIRMWELL, the first framework that models firmware rehosting as coordinated emulation of both the target service and its user-space dependencies. FIRMWELL follows a key principle: **rely on the firmware's own initialization logic to create dependencies whenever possible, while precisely identifying and fixing underlying system-level emulation errors**. FIRMWELL operates in two phases. First, it performs dependency-aware rehosting by establishing a multi-process emulation environment and executing the complete *init routine* before launching the target service. To avoid introducing additional errors, FIRMWELL employs conservative initial emulation strategies for system-level resources. Second, FIRMWELL iteratively identifies and solves rehosting failures if there is any. Given a failure, it locates the specific process blocking successful rehosting in the process tree by tracing the target's execution dependencies through call-chain analysis. Then, based on system call patterns and error symptoms, the exact failure is identified and classified as either user-space dependency issues or system-level emulation errors. For user-space issues, it either relocates misplaced resources or traces back to the peer process responsible for initialization, ultimately identifying the underlying system-level error. For system-level errors, FIRMWELL applies progressive fixes: starting with simple strategies like creating dummy device files, and escalating to program analysis techniques (def-use analysis and symbolic execution) when needed. This iterative refinement continues until successful rehosting or reaching a retry limit.

FIRMWELL is evaluated on a dataset containing 14,049 Linux-based firmware images covering 13 major vendors. Generally, it shows a clear advantage even compared to the union results of all state-of-the-art rehosting techniques: it successfully rehosted web servers of 6,490 (46%) firmware image with functionality correctness, which is 1.8x compared to FIRMAE's 3,581 (25%) and 1.6x to GREENHOUSE's 3,962 (28%), and 8x compared to PANDAWAN's 810 (6%) services. Notably, only 563 (4%) servers are uniquely rehosted by either FIRMAE, GREENHOUSE or PANDAWAN while 1,389 (9%) are uniquely rehosted by FIRMWELL. When evaluated on rehosting other network services such as UPnP and DNS, FIRMWELL shows even more significant effectiveness advantages. As for efficiency, on average, FIRMWELL reducing the average rehosting time by 1.8 to 8.4-fold (from 12 to 22, 74, and 101 minutes). Further analysis indicates that the performance superiority of FIRMWELL mainly sources from its better preparation of the user-space dependencies and ablation study proves that all key designs of FIRMWELL contributes to its effectiveness significantly. Besides, FIRMWELL has been applied to support real-world firmware fuzzing at scale. In total, 1,043 firmware has been fuzzed using FIRMWELL emulated environments, with 67 zero-day vulnerabilities found and 10 CVEs assigned.

In summary, our contributions are:

- We identified the limitations of existing firmware rehosting approaches, which inadequately leverage user-space knowledge by failing to ensure complete execution of *init routine*, and applying superficial fixes to user-space emulation errors.
- We proposed FIRMWELL, the first framework that models firmware rehosting as coordinated emulation of both the target binary and its dependencies, with a principled approach that maximizes utilization of user-space dependency initialization logic while precisely fixing system-level emulation errors.
- We demonstrated through comprehensive evaluation on 14,049 firmware images that FIRMWELL outperforms state-of-the-art tools by 1.6-8x in rehosting success rate while reducing rehosting time by up to 8.4x.
- We validated FIRMWELL's practical impact by discovering 67 zero-day vulnerabilities with 10 CVEs assigned, demonstrating its effectiveness for security analysis at scale.

To facilitate future research, we have released the source code of FIRMWELL and the evaluation results at https://github.com/qc9c/FIRMWELL.

II. Preliminaries

A. Background

Linux-Based Firmware Rehosting Rehosting enables firmware execution in virtualized or emulated environments rather than on physical hardware, facilitating security analysis, debugging, and security testing. This paper focuses on rehosting user-space services in Linux-based firmware, particularly network services like HTTP servers in routers which doesn't heavily rely on functionalities from specific hardware peripherals. Existing QEMU-based approaches [9] employ two main strategies: full-system emulation [10], [6], [7], [11], [12], which replaces the original kernel with a customized, bootable kernel to provide complete user-space functionality, and user-mode emulation [8], which only emulates system-level interfaces required by the target application.

System-Level Functionality This comprises kernel functionalities and hardware peripherals. The former covers process control mechanisms and the proc filesystem (`procfs`), which provide necessary information and control over running processes, while the latter has hardware-specific components that enable device operations and interactions. The fundamental challenge in rehosting roots in system-level functionality discrepancies between physical devices and their emulated counterparts. Vendor-customized Linux kernels often have unique configurations and peripheral models that are not present in emulated kernels. Without precise emulation of these functionalities, user-space programs may crash or exhibit degraded functionality. Perfect emulation of hardware and kernel modules would enable seamless execution of any user-space program.

User-Space Dependency User-space dependencies encompass configurations established by *init routine*: the first user-space process (PID 1) that initializes the operating environment. This includes execution configurations (EXE), such as command-line arguments, working directories, and environment variables; dynamically generated files (DYN); and inter-process communication channels (IPC). The *init routine* performs critical setup tasks such as initializing peripherals, configuring networks, establishing system environments, and launching services, which is a complex procedure involving hundreds of processes. Its extensive system-level interactions make it particularly sensitive to emulation inaccuracies. When the *init routine* fails to complete due to imperfect emulation, essential user-space dependencies can remain uninitialized, causing subsequent rehosting failures.

B. Motivation Example

Figure 3 presents a simplified firmware startup sequence that demonstrates the limitations of existing rehosting approaches. Three programs execute sequentially: `/sbin/rc` (the *init routine*), `/sbin/http_manager` (a daemon service), and `/usr/sbin/httpd` (the target web server). `rc` configures the system environment and launches services, including `http_manager`. `http_manager` creates shared memory resources and then spawns `httpd`, the rehosting target

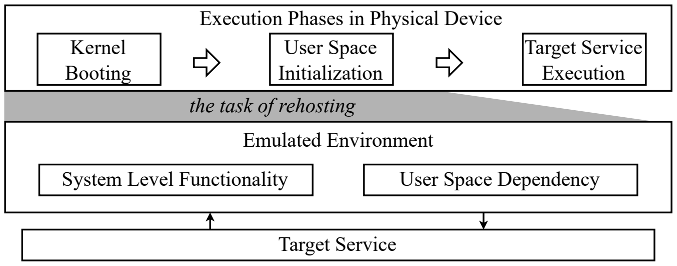


Fig. 2: The Task of Rehosting User-Space Services.

init routine -- /sbin/rc (1st executed)

```
 1 fd = open("/proc/sys/net/ipv4/dns_hijack", O_RDONLY);
 2 data = read(fd, buf, buf_size);
 3 if (data) // E1: crash, null pointer dereference as
             accessing non-exist procfs entry
 4 ...
 5 // generate config file "/etc/http.conf" for httpd service
 6 ...
 7 system("/sbin/hotplug"); // E2: endlessly stuck
 8 ...
 9 start_service("/sbin/http_manager -d");
```

init routine -- /sbin/http_manager (2nd executed)

```
10 read(open("/dev/flash", O_RDONLY), dev_buf, buf_size);
11 if (dev_buf != NULL || ! strncmp(dev_buf, "EMMC",4))exit(-1);
12 // set shared memory
13 shm_id = shmget(123, size, IPC_CREAT|0666);
14
15 system("/usr/sbin/httpd -d /www -c **.cgi &");
16 ...
17 // E3: if patch line 11, dev_buf is NULL
18 while (1){
19     ipc_communicate(shm_id, dev_buf, ...);
20 }
```

target service -- /usr/sbin/httpd (3rd executed)

```
21 // parse command-line arguments
22 // parse config file "/etc/http.conf"
23 ...
24 shm_id = shmget(123, 0, 0666); // read shared memory
25 if (shmid == -1) exit(-1); // E4: abort, the daemon
                                 service did not start
26
27 while (1) { // main service loop
28     ipc_communicate(shm_id, ...);
29     ...
30 }
```

Fig. 3: Example Code Simplified From Real-World Cases.

service. Once started, `httpd` retrieves the shared memory identifier and uses it to communicate with `http_manager` during its main service loop, which is a critical dependency for proper web server functionality. While these programs execute correctly on physical devices, they fail in emulated environments due to missing system-level functionalities, which are marked as errors `E1` - `E4` in the figure.

M1: Target Services Require Complete Initialization Errors `E1` and `E2` demonstrate how system-level emulation failures in the *init routine* cascade into missing user-space dependencies. `E1` occurs when accessing the non-existent `procfs` entry `dns_hijack`, causing a null pointer dereference. `E2` results from `hotplug` waiting indefinitely for unemulated peripherals. Although these errors appear unrelated to the target `httpd`, they prevent the *init routine* from completing its execution. Consequently, `http_manager` never start, the shared memory never be created, and `httpd` fails at line 24-25 when attempting to access this missing

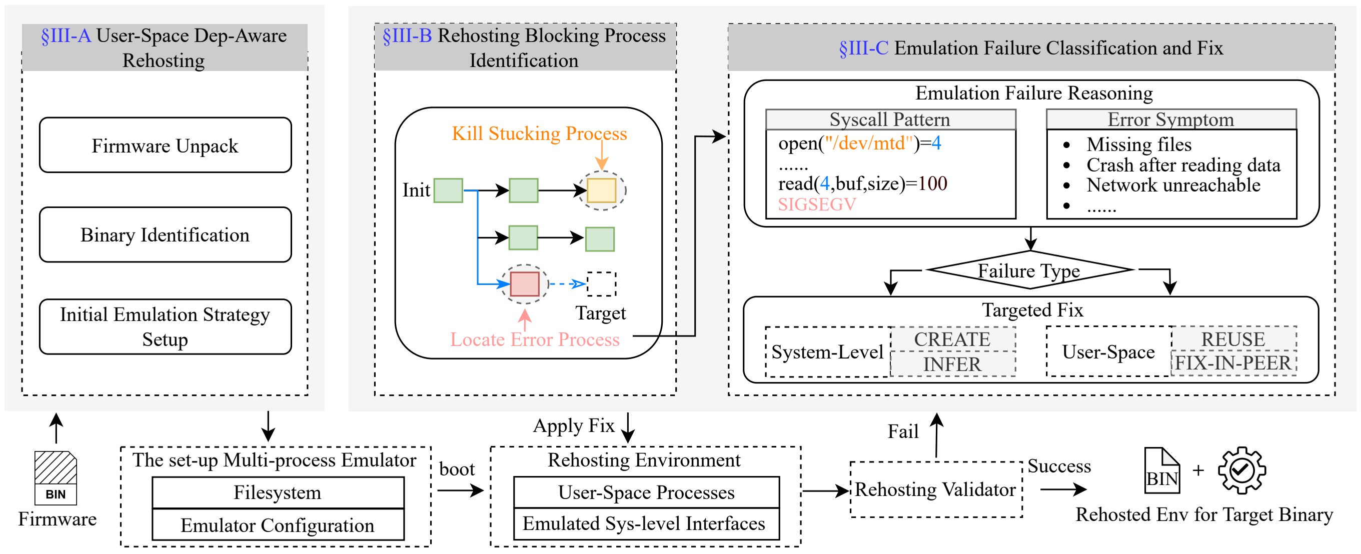


Fig. 4: Overview of FIRMWELL.

resource. Existing approaches lack mechanisms to handle these emulation failures: system-mode assumes the *init routine* executes error-free while user-mode overlooks it entirely, both failing to establish critical dependencies. This motivates us to develop a systematic mechanism that rehosts both the target service and its complete *init routine* by properly handling emulation failures throughout the dependency chain.

M2: Dependency Failures Trace to System-Level Errors Error `E4` illustrates why fixing user-space dependency failures requires addressing their system-level root causes. When `httpd` fails to access shared memory, the immediate cause is that `http_manager` never initializes it. However, `http_manager` itself failed earlier due to incorrect emulation of `/dev/flash` (line 10). Current approaches attempt to fix `E4` directly by creating shared memory, but this fails because the IPC channel also requires active communication from `http_manager`. Similarly, force patches like bypassing the device check (line 11) still fail because `dev_buf` remains null, breaking IPC functionality (line 19). The fundamental issue is that user-space dependency failures are symptoms of underlying system-level emulation problems. Effective rehosting must trace these dependency failures back to their system-level causes and provide semantically correct emulation, such as inferring valid device values that enable proper IPC communication. This motivates us to develop an approach that relies on the firmware's own initialization logic to create dependencies whenever possible, while employing program analysis to precisely infer system-level resource values when emulation errors occur.

C. Overview

Figure 4 illustrates FIRMWELL's two-phase workflow: dependency-aware rehosting followed by iterative failure resolution. In the first phase, FIRMWELL establishes a multi-process emulation environment to execute both the *init routine* and the target service. After unpacking the firmware and identifying the initialization entry point, FIRMWELL applies conservative emulation strategies for system-level resources to avoid introducing additional errors while allowing the firmware's own logic to establish user-space dependencies (Section III-A). When emulation failures occur, FIRMWELL enters its iterative resolution phase. It first identifies the blocking process by correlating runtime process tree information with static call-chain analysis (Section III-B). Based on system call patterns and error symptoms, FIRMWELL classifies each failure as either a user-space dependency issue or a system-level emulation error (Section III-C). For user-space dependency failures, e.g., missing files, broken IPC, FIRMWELL either relocates the files already existed in the environment (`REUSE`) or traces back to the responsible initialization process to uncover the underlying system-level error (`FIX-IN-PEER`). For system-level emulation errors, FIRMWELL applies progressive fixes: from creating dummy resources (`CREATE`) to employing program analysis techniques that infer semantically correct values (`INFER`). This iterative process continues until the target service successfully rehosts or reaches the retry limit.

III. METHODOLOGY

A. User-Space Dependency-Aware Rehosting

Multi-Process Emulator To ensure complete user-space dependency construction during emulation, FIRMWELL requires an emulator capable of executing multiple processes with proper resource sharing and IPC support. While QEMU-system mode natively provides this functionality, it incurs significant overhead from full VM (Virtual Machine) emulation—approximately 26x slower than user-mode according to prior work [12]. For performance considerations, we construct a container-based multi-process emulator built on QEMU-user mode.

Our approach leverages containers to create VM-like emulation environment while avoiding VM emulation cost. Although QEMU-user mode can support emulating multiple processes with shared resources and proper inheritance by launching multiple QEMU-user mode instances, specific design is required to ensure correct emulation resource sharing and isolation within containers. Our isolation strategy comprises:

- ***init routine* takes PID 1** Each firmware runs in a dedicated container where all processes share resources, with the *init routine* set as entrypoint to ensure it runs as PID 1.
- **Mock procfs** We redirect `/proc` accesses to a writable mock path (`/fakeproc`) via hooking file-access-related syscalls, which can support sequential `procfs` item read/write operations. This helps in both runtime system-level emulation adjustment and isolation between concurrent containers.
- **Enforcing binary interpreter for QEMU user-mode** We patch QEMU-user's `execve` syscall to ensure spawned processes use the container-internal QEMU user-mode binaries rather than the host's interpreter (by forcing the `path` argument of `execve(path, argv, envp)` to be `qemu_user_mode_path`). By default, QEMU user-mode hasn't specify the interpreter when spawning subprocesses, which means the interpreter will be determined by kernel feature `binfmt_misc`[2]. This will bypass container filesystem isolation, causing the host QEMU user-mode be executed with host resources access, leading to race issues when concurrently launching multiple rehosting containers.
- **Filesystem and Device Isolation** Hardware peripherals, system files, user-space files are either emulated before or during rehosting or extracted from initial firmware image. They are treated as regular files within the container, while network devices and `devfs` utilize container-native configurations.

Despite these enhancements, QEMU-user mode occasionally encounters compatibility limitations, such as unimplemented `clone` flags [14]. When FIRMWELL detects such issues, it automatically falls back to QEMU-system mode with our independent implementation that provides initial hardware configuration at startup and fixes emulation failures iteratively, either through direct file modifications or via a custom kernel that intercepts system calls (e.g., `ioctl`) to emulate missing peripherals. This dual-mode design balances performance and compatibility, enabling efficient rehosting across diverse firmware images.

Conservative Initial Emulation Strategy For user-space dependencies, it does nothing but rehosts the *init routine* before rehosting the target. For system-level functionalities, unlike existing approaches [6], [7] that pre-populate device files based on hardcoded lists, e.g., `/dev/gpio`, FIRMWELL adopts a conservative initialization strategy. We provide only the minimal resources required for basic operations, allowing the firmware's own initialization logic to create additional dependencies as needed. This approach intentionally exposes firmware-specific requirements rather than masking them with potentially incorrect default emulation heuristics, enabling more precise fixes during the iterative resolution phase.

- **procfs & devfs** FIRMWELL creates only essential device files necessary for boot (`/dev/console`, `/dev/random`, `/dev/urandom`, `/dev/tty`), while `procfs` inherits the container's initial state with the *init routine* as PID 1. Additional devices are created on-demand when rehosting failures reveal specific requirements.
- **NVRAM** NVRAM (Non-Volatile Random Access Memory) enables firmware to persist configuration across reboots [6]. FIRMWELL interposes NVRAM operations through `LD_PRELOAD` to provide a key-value store that returns empty strings for unset keys, preventing crashes while enabling subsequent value inference. (see Section IV for details).
- **Network Configuration** A default bridge interface is set to provide basic network connectivity.

Rehosting Validation When no new processes are detected within the specified timeout period, FIRMWELL concludes that rehosting is finished and proceeds to validate the rehosting outcome. Similar as existing works [8], FIRMWELL adopts the manual configured functionality validation criteria to check if the expected functionality is provided. For instance, rehosting an HTTP service is considered successful if interactions with the rehosted server return normal status codes and no predefined error strings. For other services, users need to setup its custom validation criteria.

B. Rehosting Blocking Process Identification

When rehosting fails during the execution of hundreds of processes in the *init routine*, FIRMWELL must identify which specific process blocks successful rehosting. Note that not all execution errors require resolution but only those that prevent the target service from launching. Blocking processes typically fall into two categories: ① processes that should initialize dependencies but exit abnormally or become stuck, and ② unrelated processes that become stuck, halting the entire *init routine*. FIRMWELL identifies the last abnormally exited process in category ① as the blocking process, while stuck processes in both categories must be addressed to continue initialization.

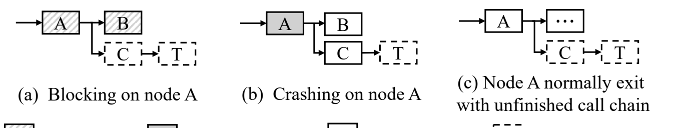
However, locating the blocking process is challenging because the *init routine* spawns numerous subprocesses, and most of them are unrelated to the target service. FIRMWELL addresses this through a two-step approach: first constructing a static call-chain graph representing the invocation paths from init to target, then mapping runtime execution records onto this graph to pinpoint the blocking process. Algorithm 1 shows the overall workflow where the dependency graph construction is at `CallChainConstruction` (line 1-14), and the blocking process identification is at `FindErrorProc` (line 15-25).

Static Analysis: Call-Chain Construction A call-chain is a directed graph $G = (N, E)$ where N represents

[2]binfmt_misc enables the kernel to invoke specific interpreters for different binary formats [13].

Algorithm 1: Error Process Localization.

```
   Input: $fs$ (filesystem of a firmware); $t$ (name of target service); $init$ (name
       of init binary); $trace$ (record of actual invocation trace of after boot)
   Output: $err\_process$ (the error process related to rehosting $t$)
1  Function CallChainConstruction(trace, target):
2      $call\_chains \leftarrow \emptyset$
3      $worklist, analyzed \leftarrow \{target\}$
4      while $worklist \neq \emptyset$ do
5          $curr \leftarrow worklist$.dequeue()
6          foreach $caller \in$ GETALLEXECUTABLE($fs$) do
7              if $curr \in$ GETSTRINGS($caller$)&$curr \notin analyzed$ then
8                  if $curr \in$ GETCALLEDSUBROUTINES($caller$) then
9                      $analyzed$.enqueue($caller$)
10                     $worklist$.add($caller$)
11                     $call\_chains$.addEdge($caller, curr$)
12     if $trace \neq \emptyset$ then
13         REFINECALLCHAIN($call\_chains, trace$)
14     return $call\_chains$
15 Function FindErrorProc(call_chains, trace, target):
16     while $True$ do
17         $n \leftarrow$ GETLASTEXECUTEDONCHAIN($call\_chain, trace$)
18         $state \leftarrow$ GETPROCESSSTATE(P)
19         if ISEXIT($state$) then
20             return $n$        // case (b) and (c) in Fig.5
21         if GETSUBPROCESS($n$) $> 0$ then
22             $proc \leftarrow$ FINDBLCOKPROCESS($n$)
23             KILLPROCESS($proc$) // case (a) in Fig.5
24             continue ; // wait and check for new processes
25         return $n$
26 if GETCALLCHAIN($target$) $= \emptyset$ then
27     $call\_chains \leftarrow$ CALLCHAINCONSTRUCTION()
28 $err\_process \leftarrow FindErrorProc(call\_chains, trace)$
```



(a) Blocking on node A (b) Crashing on node A (c) Node A normally exit with unfinished call chain

▨ blocking node ▩ abnormal exit node □ normal exit node ⬚ non-executed node

Fig. 5: Examples of Error Processes (A node represents an executable file, such as a shell script or ELF binary) .

executables (including shell scripts and ELF binaries) and E represents invocation relationships through `execve`-like calls. Computing the complete call-chain starting from init would require analyzing hundreds of binaries with high computational cost. Instead, FIRMWELL employs backward analysis starting from the target, effectively pruning irrelevant execution paths (lines 1-16).

The construction proceeds in two phases. First, for each executable in the worklist, FIRMWELL identifies potential parents by searching for the executable's path in all firmware binaries' string constants (line 7). This is a quick but rough process to filter out the candidate caller binaries. Second, candidates are verified through Def-Use analysis to confirm actual invocations by checking if the target appears in called subroutines (line 8). Verified parents are added to the worklist for recursive analysis (lines 9-10), and confirmed invocations create edges in the call-chain (line 11). This targeted approach reduces analysis scope from hundreds to dozens of binaries: only those on the actual invocation path to the target. Besides, FIRMWELL will also utilize runtime trace data to further collect the invocation relationships missed by static analysis and use the enhanced invocation set to identify missed call-chains.

Dynamic Analysis: Blocking Point Identification With the call-chain established, FIRMWELL maps runtime execution records to identify where initialization stalls (lines 15-25). The key idea is that the *init routine* executes all programs in the call-chain sequentially and the last executed node on the call-chain is the furthest point reached before failure.

After identifying the last executed binary, FIRMWELL finds its process in the process tree and analyzes this process's state (line 19). FIRMWELL divides it as three scenarios as shown in Figure 5: ① If the process remains running with active children, FIRMWELL identifies all child processes and terminates the deepest one in the recursion (lines 21-23), then continues initialization (line 24). If there is no subprocess, the process itself will be identified as blocking process; ② If the process exited abnormally, it is marked as the blocking process (line 20); ③ If the process exited normally but failed to invoke its successor, it is also marked as blocking. This iterative process continues until the rehosting blocking process is identified, and that process will be further analyzed to propose the proper fix strategies in the next section.

C. Emulation Failure Classification and Fix

Overview After identifying the blocking process, FIRMWELL applies targeted fix strategies based on error classification. FIRMWELL distinguishes between two fundamental error types: system-level emulation errors, e.g., missing device files, incorrect peripheral responses, and user-space dependency errors, e.g., missing configuration files, broken IPC channels. The key insight is that in a functioning firmware, user-space errors occur only when system-level emulation failures prevent proper initialization. Thus, FIRMWELL fixes system-level errors directly through improved emulation, while resolving user-space errors by tracing back to and fixing their underlying system-level causes. Once the actual cause is addressed, the user-space error resolves automatically.

Table I presents our complete error taxonomy and corresponding fix strategies. For system-level errors, FIRMWELL distinguishes between missing resources (handled by CREATE through creating dummy emulation resources) and incorrect emulation (handled by INFER through inferring correct values using program analysis). This progressive approach, i.e., starting with simple dummy resources before applying complex inference, follows the same conservative principle as our initial emulation strategy: find the simplest emulation that enables successful rehosting while minimizing the risk of introducing additional errors or unnecessary complexity. For user-space errors, FIRMWELL only addresses inter-process scenarios, as we have not observed intra-process cases in practice (their theoretical solutions are discussed in Appendix A-E). Errors are further classified as either misplaced dependencies (files exist but in wrong locations, fixed by REUSE) or missing dependencies (never properly created due to initialization failure, addressed by FIX-IN-PEER which identifies the responsible peer process and fixes its system-level errors).

Nonexistent System-Level Functionality Access ➔CREATE

- *Network Configuration* When network services fail to bind to specific devices or IP addresses absent

TABLE I: Emulation Failure Categories.

Categories of Misemulation	Classification Condition (Involved Syscall Pattern+Error Symptom)	Fix Strategy
Misplaced Inter-Process User-Space Dependencies (user-space files)	1) file-access syscalls failed + the file can be located in the filesystem	REUSE
Nonexistent or Incorrectly Emulated Inter-Process User-Space Dependencies (user-space files, IPC)	1) file-access syscalls failed + the file can't be located in filesystem 2) `shmget` / `connect` failed + peer processes not established 3) `shmget` / `connect` / `open` successed + data-read successed + process crashed	FIX-IN-PEER
Nonexistent System-Level Functionality Access (devfs, Network)	1) file access syscalls failed 2) `bind`/`setsocket`/`ioctl(fd, SIO*, ...)` failed + network unreachable	CREATE
Incorrect Emulation of System-Level Functionality (Peripheral, devfs, procfs, NVRAM)	1) `open` NVRAM files failed* 2) `ioctl` failed + process crashed 3) data-read–like syscalls successed + process crashed	INFER

* We modify NVRAM key-value pair access to file content, with key access logged by the `open` syscall.

in the initial environment, FIRMWELL creates the required network resources. Upon detecting errors from network-related syscalls (`bind`, `setsockopt`, `ioctl`), FIRMWELL configures a network bridge and assigns dummy network interfaces with appropriate IP addresses, establishing connectivity between host and guest.

- *Device and System-Level Files* For missing `devfs` and `procfs` entries, FIRMWELL creates files based on the failed access patterns. When `open`-like syscall fails, an empty regular file suffices to bypass existence checks. When `stat` fails, FIRMWELL creates the appropriate file type (character or block device) inferred from the filename pattern, ensuring correct metadata for type-sensitive operations.

Incorrect Emulation of System-Level Functionality →INFER When CREATE's empty files prove insufficient, INFER provides semantically meaningful values for data-read operations. FIRMWELL employs a three-tier progression: random values, database lookup, and program analysis, escalating only when simpler approaches fail. This strategy is built based on the observation that many programs only check value existence rather than content, minimizing both analysis overhead and the risk of incorrect emulation.

- *Random Value Generation* FIRMWELL first attempts random values for system-level resources' read operations. Device files like `/dev/mtd` are populated with 256MB of random data (exceeding typical embedded device memory) to satisfy any access pattern. For `ioctl` calls, FIRMWELL identifies the resource via `fd` and `request` parameters, then returns fixed values, e.g., `0xdeadbeef` and success status.
- *NVRAM Configuration Inference* When random values fail for NVRAM entries, FIRMWELL applies targeted inference. First, it searches a pre-built database of vendor-specific key-value pairs extracted from factory reset paths, e.g., `/etc/default` [6], [7]. If the key remains unresolved, FIRMWELL performs intra-procedural def-use analysis to extract values from the binary itself: ① tracking constants passed to `nvram_set`, and ② identifying literals compared against `nvram_get` return values via comparison functions such as `strcmp`. This approach leverages NVRAM's string-based nature to provide semantically correct configurations.
- *Magic Byte Inference via Symbolic Execution* When firmware verifies specific byte patterns, e.g., version numbers at fixed offsets in `/dev/flash`, random values and def-use analysis often fail to capture these constraints. To address this, FIRMWELL maps execution traces to the binary control flow graph, locates abnormal exit calls or crash instructions, and then launches targeted symbolic execution from the identified error point to find satisfying constraints. Specifically, FIRMWELL initializes the relevant system-level resource as symbolic values, handling both file contents and `ioctl` syscalls, and executes to find satisfying constraints. To manage complexity, analysis is scoped to the current function that accesses the system-level resource, with concrete values for existing resources and zero-filled uninitialized data. This error-driven approach efficiently extracts magic bytes and similar fixed-pattern constraints that would otherwise block rehosting.

Misplaced Inter-Process User-Space Dependencies →REUSE The REUSE strategy handles cases where required files exist in the firmware but remain inaccessible due to incomplete initialization. FIRMWELL searches for these files in typical service directories, e.g., `/www`, and applies two resolution approaches: ① for relative path failures, it changes the working directory via `chdir`; ② for absolute path failures, it copies files to their expected locations. This simple relocation often suffices when the dependency itself is intact but merely misplaced.

Missing Inter-Process User-Space Dependencies →FIX-IN-PEER When user-space dependencies cannot be relocated (REUSE), FIRMWELL traces the failure to its source: a peer process that failed to initialize the resource due to system-level emulation errors. Unlike system-level resources that can be directly emulated, these dependencies embed process-specific semantics, e.g., IPC protocols, data formats, file contents, that require the original initialization logic.

FIRMWELL identifies the responsible peer process through three steps: ① It first extract the IPC paradigm and its associated data key [3] from the syscall traces of the current blocking process. FIRMWELL then performs initial filtering by searching these identifiers as byte sequences across all firmware binaries to quickly identify peer candidates. ② It refines candidates using lightweight symbolic execution to verify actual IPC usage. Starting from the entry points of functions that call IPC-related functions, e.g., `bind`, `connect`, `shmget`, FIRMWELL symbolically executes to extract argument values and confirm they match the resource accessed by the blocking process. This step did a precise filtering which can even filter out binaries that merely contain the byte sequence but don't actually interact with the resource. ③ For each verified peer process, FIRMWELL applies the complete error diagnosis and fix workflow described in this section. Based on our empirical observations, Unix domain sockets, shared memory, and user-space files are the primary IPC mechanisms that fall into this failure category. Note that, while multiple candidates could theoretically be identified, requiring either exhaustive testing of all candidates or more sophisticated setter/getter identification techniques, we have rarely encountered such cases in our evaluation.

IV. IMPLEMENTATION

We implemented a prototype of FIRMWELL with 27,646 LoC of Python code and 414 lines of Bash scripts. The system begins with a firmware pre-processing stage, which extracts the root filesystem and identifies key binaries for subsequent analysis and emulation. FIRMWELL has three key components: *Multi-Process Emulator*, *Blocking Process Identifier* and *Emulation Failure Fix*. Due to the page limit, we cannot detail every engineering detail in this section. Instead, we discussed several interesting implementation design choices about FIRMWELL's rehosting framework.

Firmware Pre-processing This stage unpacks the firmware to extract the filesystem, including executables and resource files, and identifies the init binary and target binary for rehosting. FIRMWELL first leverages Binwalk [16] for initial unpacking. If Binwalk fails, FIRMWELL uses FACT [17], which leverages community plugins to improve extraction coverage [18]. Since the default FACT workflow does not retain the original directory structure, which is critical for correct rehosting, we developed a customized FACT version that preserves the complete hierarchy during extraction. Following prior work [6], we treat extraction as successful if at least four directories defined in the Filesystem Hierarchy Standard (FHS), e.g., `/bin`, `/etc`, `/lib`, and `/usr`, are identified. We empirically expanded the set of recognized *init routine* binaries based on large-scale observations. Detailed identification procedures for both the *init routine* and target binaries are provided in Appendix A-A.

[3] A data key refers to the unique identifier used by communicating processes to access a specific IPC resource, such as a socket path, shared memory key or file path [15]. For instance, in socket-based IPC, the data key can be the file path of a Unix-domain socket or the combination of IP address and port specified in address structures (e.g., `sockaddr_in`).

Syscall Execution Tracer To capture execution context and dependency relationships during rehosting, we developed a lightweight tracer based on syscall instrumentation. For QEMU-user mode (v8.20), we patched 219 lines of code to intercept syscall invocations and log execution information. Specifically, FIRMWELL: ① hooks the `execve` syscall to record program launches, capturing command-line arguments and environment variables. ② hooks file-access-related syscalls, such as `open` and `access`, to redirect file paths and handle missing `procfs` entries. ③ hooks the `bind`, `setsockopt`, and `ioctl` syscalls to record peripherals-related operations. ④ hooks IPC-related syscalls such as `bind` and `connect` to record IPC communications. A similar syscall hooking mechanism is implemented on a customized Linux kernel (v4.1) to support execution tracing in QEMU-system mode.

libnvram.so To support NVRAM emulation, FIRMWELL provides a custom `libnvram.so` library (854 lines of C code) that intercepts and log all access operations using `LD_PRELOAD`. This library acts as a key-value database, handling `nvram_set` to update values and `nvram_get` to retrieve them. If a requested key has not been set, the library returns an empty string by default. Otherwise, it returns the stored value. This is similar as prior works [6], [7], [8].

Details of Call-Chain Construction Given each executable, FIRMWELL extract its invoked subprocesses through the following analyses, which serve as the foundation for call-chain construction. For ELF binaries, FIRMWELL performs intra-procedural Def-Use analysis to recover the arguments passed to `execve`-like API call sites. For shell scripts, FIRMWELL constructs the abstract syntax tree (AST) based on Morbig [19], and then traverses command-related AST nodes to identify all subprocess invocations.

Def-Use Analysis FIRMWELL leverage intra-procedural binary analysis to extract constant strings propagated to specific function callsites. The analysis is implemented using Ghidra's P-code intermediate representation [20].

Emulation Failure Classification and Fix We leverage angr [21] as the symbolic execution engine for: ① *magic byte inference*, FIRMWELL synthesizes concrete values for system-level resources in INFER fix strategy. The analysis is restricted to the current function accessing the resource, skipping all callees for simplicity. FIRMWELL provide concrete values for existing files and zero-initialize unconstrained memory and registers, i.e., angr's `ZERO_FILL_UNCONSTRAINED_MEMORY` option. ② *peer process identification*, FIRMWELL extracts the IPC identifier, *i,e, data key*, associated with the specific IPC paradigm. The extracted identifier can include string values and binary representations of structures like `sin_addr` within `sockaddr_in`. Besides, in this work, we set the maximum number of fix iterations to 10 for one process, and the maximum number of processes analyzed within a firmware to 5.

TABLE II: Number of Web Servers Rehosted by FIRMAE, GREENHOUSE, and FIRMWELL. The number of Unpack is from FIRMWELL.

Brand	# of Images	Unpack	FirmAE			Greenhouse			Pandawan			FIRMWELL		
			Execute	Connect	Interact	Execute	Connect	Interact	Execute	Connect	Interact	Execute	Connect	Interact
ASUS	1,902	1,701	1,716	764	107	1,484	1,300	1,284	77	35	30	1,585	1,585	**1,579**
AVM	524	0	291	1	**1**	0	0	0	0	0	0	0	0	0
Belkin	63	63	57	23	8	61	47	**34**	24	10	0	38	38	32
D-Link	3,316	1,966	2,289	1,015	950	1,558	975	872	470	289	277	1,485	1,134	**1,116**
EDIMAX	200	159	146	46	**46**	121	28	18	2	0	0	24	24	19
Engenius	143	70	128	36	34	66	59	59	3	1	1	69	69	**69**
Linksys	313	241	234	152	125	228	72	67	42	25	7	182	149	**136**
Netgear	3,421	2,663	2,954	1,612	1,320	2,549	1,486	1,029	495	253	210	2,462	2,305	**2,251**
Tenda	172	153	161	15	9	138	69	**60**	41	0	0	68	48	48
TP-Link	1,637	1,279	1,375	746	650	1,137	264	250	548	318	269	859	859	**865**
TRENDnet	967	574	805	374	324	612	352	271	70	19	16	486	397	**363**
Ubiquiti	1,377	643	1,192	1	1	437	12	**12**	28	0	0	1	1	1
Zyxel	20	19	19	8	6	18	7	6	3	0	0	16	16	**11**
Total	14,049	9,531	11,367	4,793	3,581	8,409	4,671	3,962	1,803	950	810	7,257	6,625	**6,490**

V. EVALUATION

Evaluation Questions The evaluation aims to answer:

- **RQ1:** How does FIRMWELL perform compared to state-of-the-art rehosting solutions?
- **RQ2:** How does each component of FIRMWELL contribute to its overall performance?
- **RQ3:** Can FIRMWELL be used to discover real-world vulnerabilities within rehosted targets?

Firmware Dataset We evaluate FIRMWELL on a dataset of 14,049 Linux-based firmware images collected from two sources: LFWC [18] (10,449 images) and GREENHOUSE [8] (7,098 images). After merging and removing duplicates using SHA256 checksums, we obtained 14,049 unique firmware images. This dataset covers 2,140 devices from 13 major vendors: ASUS, AVM, Belkin, D-Link, EDIMAX, EnGenius, Linksys, Netgear, Tenda, TP-Link, TRENDnet, Ubiquiti, and Zyxel. Device types include routers, cameras, switches, and wireless access points, etc. Note that we use FIRMWELL's firmware unpacking workflow to extract the filesystem from each firmware image, which is then used as the input for analysis by all tools.

Configurations All experiments were conducted on a Kubernetes cluster with 1,000 CPU cores. Each pod was allocated 1 CPU core and 4GB of RAM, except for PANDAWAN pods, which received 4 CPU cores per their recommended configuration. The maximum rehosting time limit was set to 12 hours.

A. *State-of-the-Art Comparison (RQ1)*

Baselines We compared FIRMWELL with three state-of-the-art rehosting tools: FIRMAE [7], GREENHOUSE [8], and PANDAWAN [22]. FIRMAE, an enhanced version of FIRMADYNE [6], uses QEMU-system mode for full-system rehosting. GREENHOUSE employs QEMU-user mode with an adaptive and iterative strategy for rehosting target services. PANDAWAN performs kernel-space rehosting through its Kernel Augmentation technique, which builds augmented kernels with vendor-specific functionality to enable holistic analysis of both user-level and kernel-level code. We excluded FIRM-AFL [23] and EQUAFL [12] as they focus primarily on improving fuzz-testing throughput for targets already rehosted by FIRMADYNE.

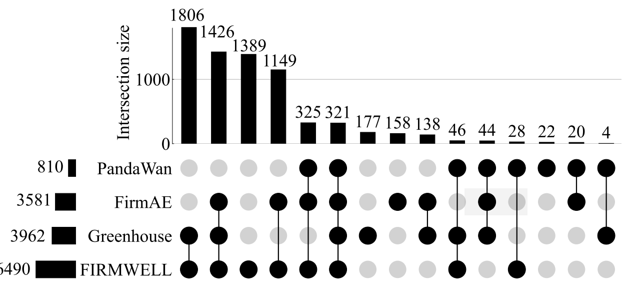


Fig. 6: The Intersection of Successfully Rehosted Web Servers Across Tools.

Evaluation Services and Metrics To demonstrate the generality of our method, we evaluated FIRMWELL on three major network services: HTTP, DNS, and UPnP. Following prior work [6], [10], [11], [12], [8], we selected HTTP as the primary target due to its prevalence and rehosting complexity.

We adopted the multi-stage criteria from Greenhouse [8]: unpacking firmware (**Unpack**), running the target service (**Execute**), connecting to the network service (**Connect**), and interacting with it to verify functionality (**Interact**). For FIRMWELL and GREENHOUSE, successful Unpack means identifying the HTTP service binary in the filesystem, while Execute is defined by detecting the `bind` syscall invocation. For full-system emulation tools (FIRMAE and PANDAWAN), Unpack means successfully extracting and mounting the filesystem as a QEMU image, while Execute indicates successful system boot with network connectivity.

Overall Results Table II presents the rehosting results for 14,049 firmware images across four tools. FIRMWELL demonstrates superior performance, successfully rehosting 6,490 firmware services to the Interact stage: **1.8×, 1.6×, and 8.0× more than FIRMAE (3,581), GREENHOUSE (3,962), and PANDAWAN (810), respectively**. Among 9,531 successfully unpacked firmware images, FIRMWELL achieved a 68% success rate overall, with notable rates of 92% for ASUS and 84% for Netgear. FIRMWELL outperformed baselines in 8 of 13 vendors, though GREENHOUSE performed better on Belkin, EDIMAX, Tenda, and Ubiquiti, while FIRMAE excelled on AVM.

Figure 6 shows the overlapping results among the four tools.

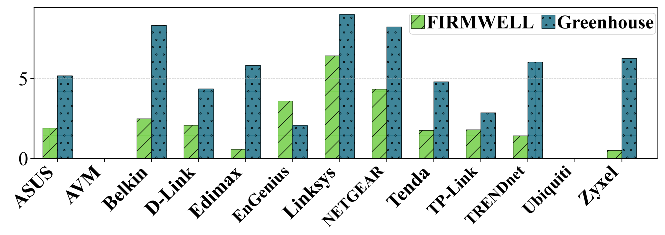


Fig. 7: Comparison of Fix Rounds between FIRMWELL and GREENHOUSE (Y-axis: Fix Rounds).

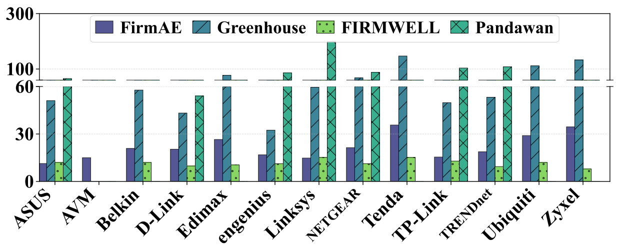


Fig. 8: Time Cost Comparison for Rehosting (Y-axis: minutes).

Of 9,531 firmware images with identified HTTP services, the tools collectively rehosted 7,053 (74%). FIRMWELL successfully rehosted 6,490 services, while the other three tools combined rehosted 5,664. **FIRMWELL uniquely rehosted 1,389 additional firmware services and covered 5,101 (90%) of the services rehosted by all baselines**.

Results on Greenhouse Dataset The results on the 7,098 firmware images from the Greenhouse dataset are provided in Appendix A-B. The evaluation confirms that FIRMWELL's performance advantages observed in the overall results remain consistent on this subset of the data.

Analysis of FIRMWELL-Only Results We analyzed the 6,490 firmware services successfully rehosted by FIRMWELL to understand its advantages in handling critical dependencies. For command-line arguments, a crucial user-space dependency, FIRMWELL obtained 522 more arguments than GREENHOUSE by executing *init routine* rather than relying solely on inference strategies. FIRMWELL's strategies also addressed two key challenges that other tools missed. First, while GREENHOUSE provides common IPC processes like `datalib` and `xmldb`, FIRMWELL's FIX-IN-PEER strategy identified additional critical IPC dependencies, including `ubusd` (required by 244 firmware images) and `cfg_manager` (needed by 165 firmware images). Second, through symbolic execution, FIRMWELL recovered system-level information such as `/proc/mtd` file contents required by 89 Netgear firmware images, where missing partition names would cause early rehosting termination due to magic byte verification failures. By effectively handling both user-space dependencies and system-level requirements, FIRMWELL achieved superior rehosting coverage.

Analysis of SOTA-Only Results We analyzed 563 firmware services successfully rehosted by other tools but not by FIRMWELL. GREENHOUSE succeeded in 356 cases where FIRMWELL failed, with 53 relying on binary patching to bypass complex peripheral verification and memory access issues. For example, in D-Link firmware that directly accesses physical memory through `/dev/mem`, GREENHOUSE's patching approach succeeded where FIRMWELL's INFER strategy proved insufficient. FIRMAE achieved 360 additional successes due to two advantages: system-mode execution that avoids QEMU-user mode compatibility issues affecting *init routine* handling (particularly for specialized systems like Linksys's sysevent-based management), and comprehensive peripheral emulation including `brcmboard`, `watchdog`, and `mtdblockX` devices through kernel modifications. PANDAWAN's 23 unique successes demonstrate the value of precise system-level interface emulation.

Analysis of Common Failures We analyzed 2,074 firmware images that all four tools failed to rehost and identified two main causes. ① **Unsupported architectures.** 510 Ubiquiti firmware images used MIPS64 architecture, which none of the evaluated tools support, including the latest QEMU-user mode. Additionally, 38 Netgear firmware images used Arctic Core architecture, and several D-Link firmware had architectures unrecognizable by the `file` command. ② **Peripheral-dependent semantics.** Certain programs read data from specific device file indices to verify firmware signatures before execution. While FIRMWELL uses error-driven symbolic execution to infer system-level content, complex signature verification algorithms cause termination upon verification failure, as observed in D-Link firmware. Furthermore, tightly coupled firmware-peripheral interactions lead to symbolic execution failures such as path explosion and constraint solving timeouts, preventing FIRMWELL from inferring required system-level content. We leave advanced automatic peripheral modeling as the future direction.

Analysis of Common Rehosted Firmware Among Four Tools Both FIRMAE, PANDAWAN, and FIRMWELL achieve rehosting by executing multiple processes initiated from the init routine. FIRMAE and PANDAWAN rely on hardcoded, manually summarized error-fix strategies, enabling them to successfully rehost 3,581 and 810 firmware images, respectively, without any need for special binary-specific handling. In contrast, FIRMWELL and GREENHOUSE both employs iterative fix strategies to address rehosting errors. To compare the efficiency of these approaches, we evaluated the number of fix iterations required across 3,599 firmware images that were successfully rehosted by both FIRMWELL and GREENHOUSE. FIRMAE and PANDAWAN were excluded from this comparison, as they do not utilize an iterative error-fixing mechanism. As shown in Figure 7, **FIRMWELL demonstrated fewer iterations compared to GREENHOUSE**. On average, FIRMWELL required 2.6 iterations while GREENHOUSE took 5.8 iterations. FIRMWELL more thoroughly prepares user-space dependencies based on the initialization procedure and performs misemulation reasoning when encountering rehosting errors. This allows FIRMWELL to focus on actual rehosting errors and perform fewer error fixes.

Time Cost Analysis We evaluated FIRMWELL's efficiency

TABLE III: Comparison of rehosting success rates between vanilla Greenhouse, Migrated_Greenhouse variants, and FIRMWELL (FW).

Venilla GH	Migrated GH (baseline)	Migrated GH w/o §III-A Sys Mode	Migrated GH w/o §III-B	FW
3,962	5,135	4,789	5,023	6,490

TABLE IV: Contribution analysis of FIRMWELL(FW)'s enhanced fix strategies compared to Migrated_Greenhouse.

FW - Migrated GH (baseline)	FW w/o IMPROVED_CREATE	FW w/o INFER	FW w/o FIX_IN_PEER
1,473 (100%)	940 (-36%)	561 (-62%)	1,024 (-30%)

TABLE V: Number of UPnP and DNS Services Rehosted by four tools. The number of Unpack is from FIRMWELL. AE=FIRMAE, GR=GREENHOUSE, FW=FIRMWELL, PD=PANDAWAN.

Brand	**UPnP**					**DNS**				
	Unpack	**AE**	**GH**	**PD**	**FW**	**Unpack**	**AE**	**GH**	**PD**	**FW**
ASUS	1,662	31	488	3	**862**	1,553	29	**1,130**	5	935
AVM	203	0	0	0	0	109	0	0	0	0
Belkin	47	0	3	0	**8**	42	15	18	0	**27**
D-Link	1,067	**533**	121	136	224	1,093	**486**	141	166	356
EDIMAX	120	**7**	6	0	6	14	**31**	9	0	5
Engenius	20	0	0	0	0	62	1	0	1	**51**
Linksys	126	**7**	4	0	2	178	51	115	2	**119**
Netgear	2,345	942	**1,335**	90	1,173	1,878	1,033	590	198	**1,172**
Tenda	106	0	**69**	0	4	47	1	**11**	0	8
TP-Link	631	13	62	8	**232**	428	91	28	15	**208**
TRENDnet	294	76	56	14	**90**	289	82	**169**	16	140
Ubiquiti	401	0	11	0	**30**	769	0	**227**	0	96
Zyxel	13	2	0	0	**3**	13	1	**5**	0	1
Total	7,035	1,611	2,155	251	**2,634**	6,475	1,821	2,443	403	**3,118**

by analyzing the average analysis time across successfully rehosted firmware services. Figure 8 shows the average analysis time cost across different firmware brands. **FIRMWELL requires only 12 minutes on average per firmware service, achieving 1.8–8.4× speedup compared to baseline approaches (22–101 minutes)**. This efficiency stems from several design choices: First, FIRMWELL implements active process monitoring and intervention, avoiding FIRMAE's fixed sleep periods (240s) and system restarts for service initialization and network reconfiguration. Second, while GREENHOUSE relies on FIRMAE's completion followed by repeated offline repairs, FIRMWELL employs targeted error localization and fixes which can more strategically handling emulation errors, minimizing the fix rounds. Finally, PANDAWAN's longer analysis time results from its comprehensive kernel module analysis and custom kernel building requirements.

Comparison with Migrated_Greenhouse To isolate the contributions of FIRMWELL's environment infrastructure from its fix strategies, we created Migrated_GREENHOUSE by integrating FIRMWELL's user-space environment infrastructure while retaining GREENHOUSE's original fix strategies. Specifically, §III-A and §III-B are applied to Migrated_GREENHOUSE for providing the same multi-process environment while GREENHOUSE's fix strategies is applied only to the target service process.

Table III shows that Migrated_GREENHOUSE achieved 29.6% higher success rate than vanilla GREENHOUSE (5,135 vs 3,962), validating the effectiveness of FIRMWELL's enhanced multi-process environment. Specifically, 346 firmware failed if removing system-mode support (§III-A) and 112 removing blocking process identification (§III-B).

Migrated_GREENHOUSE uniquely succeeded in 118 cases due to GREENHOUSE's PATCH strategy, which bypasses version checks in certain D-Link firmware where INFER alone fails. However, FIRMWELL excludes PATCH to preserve user-space dependency semantics, as aggressive patching can cause unexpected side-effects. An experimental version (FIRMWELL_GH) that applies PATCH when standard rehosting fails yielded only 225 additional successes, suggesting unique but limited benefits.

We analyzed 1,473 firmware images where FIRMWELL succeeded but Migrated_GREENHOUSE failed. FIRMWELL's fix strategies improve upon GREENHOUSE's while maintaining compatibility with its REUSE strategy: ① eliminates PATCH to preserve binary integrity; ② enhances CREATE with comprehensive device support including char devices and improved network device error detection (e.g., SIO* flags in `ioctl`); ③ introduces INFER and FIX_IN_PEER strategies. Table IV quantifies these improvements. FIRMWELL's enhanced CREATE handled 36% (533/1,743) more cases through comprehensive device emulation. INFER resolved 62% (912/1,473) of failures, while FIX_IN_PEER addressed 30% (449/1,473) by fixing peer process issues and restoring IPC communication.

Rehosting of Other Services We evaluated FIRMWELL's performance on two additional network services: UPnP and DNS. For functionality verification, we sent protocol-specific messages (UDP-based UPnP messages and DNS queries) and parse responses to confirm service operation.

Table V presents the results. For UPnP services, FIRMWELL successfully rehosted 2,634 out of 7,035 services (37.4%), outperforming FIRMAE (1,611, 22.9%), GREENHOUSE (2,155, 30.6%), and PANDAWAN (251, 3.6%). For DNS services, FIRMWELL achieved even better results, rehosting 3,118 out of 6,475 services (48.1%), compared to FIRMAE (1,821, 28.1%), GREENHOUSE (2,443, 37.7%), and PANDAWAN (403, 6.2%). These results demonstrate that **FIRMWELL's approach generalizes to other firmware services while maintaining performance superiority**. We discuss limitations and broader applicability in Section VI.

B. Ablation Study (RQ2)

Effectiveness Analysis Per Module To understand the contribution of each key module in FIRMWELL, we performed an ablation study. Table VI presents the results. Using only the initial rehosting environment (§III-A), FIRMWELL successfully rehosted 2,239 out of 6,490 services (31%). For the remaining 4,251 services, we analyzed the impact of disabling modules §III-B and §III-C. Disabling §III-B (*Rehosting Failure Process Identification*) reduced successful rehostings to 5,920 (91%), demonstrating its critical role in identifying and terminating blocking processes. Similarly,

TABLE VI: Analysis of the contribution of three components (§III-A, §III-B, §III-C) and four fix strategies of FIRMWELL.

Brand	**ASUS**	**Belkin**	**D-Link**	**EDIMAX**	**EnGenius**	**Linksys**	**NETGEAR**	**Tenda**	**TP-Link**	**TRENDnet**	**Ubiquiti**	**Zyxel**	**Total**
§III-A only	400	574	188	201	52	60	6	5	7	4	10	0	2,239 (-66%)
w/o §III-B	1,551	26	1,034	19	66	121	1,965	46	736	344	1	11	5,920 (-9%)
w/o §III-C	1,037	14	592	11	8	71	689	41	746	273	0	11	3,493 (-46%)
w/o create	1,452	31	651	17	63	117	988	44	767	325	1	11	4,467 (-31%)
w/o fix_in_peer	1,373	31	1,009	13	68	120	1,968	45	727	338	1	10	5,703 (-12%)
w/o infer	1,311	27	968	18	64	108	1,197	46	746	336	1	11	4,833 (-26%)
w/o reuse	1,170	18	978	19	10	83	1,958	46	714	316	0	11	5,323 (-18%)
Total	1,579	32	1,116	19	69	136	2,251	48	865	363	1	11	6,490 (100%)

disabling §III-C (*Emulation Failure Classification and Fix*) resulted in only 3,493 (54%) successful rehostings, confirming its importance for classifying emulation error and applying appropriate emulation strategies.

Effectiveness Analysis Per Fix Strategy To understand the contribution of each fix strategy in FIRMWELL, we performed an ablation study by disabling one strategy at a time and rerunning the HTTP service rehosting. Table VI shows the breakdown for 6,490 firmware images requiring fix strategies. Disabling CREATE reduced success to 68% (4,467), demonstrating its critical role. Our analysis reveals that creating dummy network devices and empty system files is essential: while their content may be irrelevant to core functionality, their presence prevents unexpected termination when programs check for their existence before accessing device data. The INFER strategy contributed significantly (4,833, 74% when disabled), as firmware programs assume physical device environments and rely on system-level interfaces for proper operation. For user-space dependencies, REUSE helped resolve file location issues (5,323, 82% when disabled), while FIX-IN-PEER addressed missing IPC paradigms when peer processes failed (5,703, 88% when disabled).

C. Real-World Application (RQ3)

Risk Assessment on N-Day Vulnerabilities To assess whether rehosted services maintain sufficient fidelity for vulnerability analysis, we tested them using N-day vulnerability proof-of-concept (PoC) code from RouterSploit [24]. Routersploit comprises a series of PoCs for N-day vulnerabilities in embedded firmware. We selected firmware services that reached the Connect stage, specifically 6,625 from FIRMWELL, 4,793 from FIRMAE, 4,671 from GREENHOUSE, and 950 from PANDAWAN, then executed 125 known N-day PoCs against each service.

Table VII summarizes the results. FIRMWELL identified 1,335 vulnerabilities: 253 password disclosures, 737 command executions, 332 information disclosures, and 13 authentication bypasses. In comparison, RouterSploit detected 954 vulnerabilities on FIRMAE, 1,010 on GREENHOUSE, and 389 on PANDAWAN. These results demonstrate that FIRMWELL maintains sufficient fidelity for effective vulnerability risk assessment.

Large-Scale Fuzzing for Vulnerability Detection We integrated FIRMWELL with fuzz-testing to detect zero-day vulnerabilities. From 7,098 firmware images provided by GREENHOUSE, we selected 1,692 latest versions. FIRMWELL successfully rehosted HTTP services to the Interact stage in 1,128 of these, which became our fuzzing targets. Large-scale fuzzing ran for several months, with each instance allocated one CPU core and a 48-hour timeout per firmware.

We used AFL++ [25] with QEMU-user mode enhanced as follows: ① Applied fork-server patches from previous work [8] to collect the `accept` function's return address as the fork point, with client connections emulated in AFL-QEMU to feed test cases. ② Generated firmware-specific seeds using Selenium [26] to automate web server interactions, capturing browser-generated network requests as valid message formats. ③ Extended AFL++'s custom mutator [25] to support common protocol formats (JSON, XML, SOAP) in HTTP request bodies for enhanced vulnerability discovery.
Table VIII shows the crashes discovered during large-scale fuzzing of 1,128 firmware images. FIRMWELL initiated fuzzing on 1,043 firmware without timeouts or crashes, triggering 24,580 raw crashes in total. Since source code-based crash deduplication tools cannot handle binary-only targets, we used `AFL-tmin` to minimize crashes and computed MD5 hashes of the minimized inputs to identify 1,582 unique crashes. Manual analysis of 48 firmware images confirmed 67 zero-day vulnerabilities: 44 null pointer dereferences, 13 buffer overflows, 6 reachable assertions, and 4 uncaught exceptions. Details are provided in Appendix A-D. We responsibly disclosed these vulnerabilities to manufacturers, resulting in 10 assigned CVE IDs.

VI. LIMITATIONS AND BROADER APPLICABILITY

Technical Stack Limitations FIRMWELL's effectiveness is constrained by the limitations of its technical stack. First, rehosting coverage is determined by the set of instruction set architectures supported by QEMU. As shown in our evaluation, firmware images based on unsupported architectures such as Arctic Core, or other proprietary formats cannot be emulated, as neither QEMU-user nor QEMU-system provides compatibility for these targets. Second, FIRMWELL's error diagnosis and value inference capabilities are constrained by current binary analysis techniques such as Ghidra [20] and angr [27]. In particular, complex firmware-peripheral interactions can lead to symbolic execution failures such as path explosion and constraint solving timeouts. For instance, certain D-Link devices like DIR-605L, frequently read values from device files like `/dev/mem` during execution, making

TABLE VII: Number of 1-day Vulnerability Detected by four tools. PD=Password Disclosure. CI=Command Injection. IL=Info Leak. AB=Authentication Bypass.

Brand	FirmAE				Greenhouse				Pandawan				FIRMWELL			
	PD	CI	IL	AB	PD	CI	IL	AB	PD	CI	IL	AB	PD	CI	IL	AB
ASUS	0	14	0	0	0	0	0	0	0	0	0	0	0	13	0	0
Belkin	5	3	0	5	6	0	0	7	0	0	0	0	0	0	0	1
D-Link	253	306	226	30	247	318	274	22	48	214	34	3	251	600	330	12
Linksys	0	3	0	0	0	9	0	0	0	0	0	0	0	16	0	0
NETGEAR	3	68	0	0	83	44	0	0	0	55	0	0	0	105	0	0
TRENDnet	13	0	25	0	0	0	0	0	12	11	12	0	2	3	2	0
Total	274	394	251	35	336	371	274	29	60	280	46	3	253	737	332	13

TABLE VIII: The number of fuzzing crashes on firmware services rehosted by FIRMWELL. FUZZ indicates a successful fuzzing run. VUL indicates the fuzzer exposed at least one crash. RAW indicates the unique crashes recorded by AFL, determined by the hash of the bitmap. Unique crash is filtering by `afl-tmin` and `md5sum`.

Brand	# of Images			# of Crashes	
	Total	FUZZ	VUL	RAW	UNQ
ASUS	148	142	72	3,876	327
Belkin	48	45	0	0	0
D-Link	165	149	41	9,872	375
Linksys	49	49	6	241	25
Netgear	248	241	57	8,865	576
Tenda	20	20	13	147	15
TP-Link	338	285	0	0	0
TRENDnet	95	95	68	1,579	264
Zyxel	17	17	0	0	0
Total	1,128	1,043	257	24,580	1,582

it infeasible for the INFER strategy to synthesize all required values and causing symbolic execution to become intractable.

Limitations of No Patch Strategy FIRMWELL deliberately excludes binary patching as an error recovery strategy, adhering to the principle of recovering user-space dependencies rather than directly modifying target code. This design decision minimizes risks of introducing additional emulation errors in multi-process rehosting. However, patching can sometimes enable additional successes by forcibly altering program control flow to bypass complex system-level validation logic, such as version or peripheral checks in some D-Link devices where the requires values are too complex to be solved by INFER strategy. Considering its limited unique contribution in firmware rehosting and inevitable side effects for potentially changing the target program's semantics, we have not incorporated it into FIRMWELL's core workflow.

Complex System-Level Functionality Emulation FIRMWELL's design philosophy favors services with peripheral-agnostic core logic, typically of network services like HTTP, DNS, and UPnP in routers, cameras, etc. Our current techniques provide dummy emulation or use program analysis for specific value inference, but cannot adequately handle complex system-level logic involving stateful functionality or dynamic hardware interactions. For firmware heavily dependent on specific peripherals, e.g., Bluetooth, USB, network drivers, FIRMWELL can prepare the user-space environment but requires additional peripheral-specific emulation for core functionality. Future work could leverage LLM-based semantic understanding to better capture binary semantics and enable more sophisticated system-level emulation.

VII. Related Work

Rehosting of Linux-based Firmware Existing techniques for rehosting Linux-based firmware are categorized into kernel-space modules [27], [22] and user-space programs [6], [10], [7], [8], [11], [12]. Kernel-space rehosting, as seen in FirmSolo [27] and Pandawan [22], improves kernel module emulation success via privilege analysis. In user-space rehosting, methods evolved from full-system emulation [6], [7] to hybrid [11] and user-mode emulation [12], [8]. Firmadyne [6] pioneered full-system emulation, while FirmAE [7] enhanced success rates through heuristics. Firm-AFL [11] combined emulation modes to improve efficiency. EQUAFL [12] migrated emulations to user-mode for better performance. Greenhouse [8] increased rehosting success by proposing iterative fix strategy to execution roadblockers in the single-service rehosting scenario. Despite these advances, there are still gaps for properly utilizing user-space knowledge to improve rehosting effectiveness, which is the main motivation of our FIRMWELL.

Rehosting of Other Type Firmware Research on rehosting primarily focuses on MCU-based firmware, where tightly coupled peripheral hardware necessitates accurate emulation for successful rehosting. P2IM [28] initially modeled peripheral behavior by analyzing access patterns. Subsequent works [29], [30], [31], [32] employed symbolic execution to infer register values, improving accuracy. SEmu [33] enhanced peripheral modeling by using specifications, especially for behaviors like interrupts. Other approaches [34], [35] utilized hardware abstraction layers (HALs) from chip vendors for rehosting, avoiding peripheral inference. However, MCU-based rehosting typically treats firmware as a single entity, limiting its direct applicability to individual Linux-based firmware binaries.

Firmware Vulnerability Analysis Firmware vulnerability detection is divided into static and dynamic analysis. Static analysis involves identifying vulnerabilities through code analysis without running the program, using techniques like taint analysis [36], [37], [38], [21], [39], [40], [41], [42] and code similarity analysis [43], [44], [45], [46], [47], [48]. However, static methods often suffer from false positives. Dynamic analysis involves executing test cases within firmware environments, utilizing either physical devices or rehosting methods. For inaccessible firmware, black-box testing on physical devices is explored [49], [50], [51], [52], [53], [54]. To enhance testing efficiency, gray-box testing through firmware rehosting is evolving [11], [12], [8]. Challenges such as incomplete user-space initialization and emulation dependency issues limit these methods, leading to insufficient testing. FIRMWELL overcomes these by improving rehosting capabilities, achieving 1.6 to 8 times more successful rehosting and discovering 67 new vulnerabilities.

VIII. CONCLUSION

We presented FIRMWELL, the first framework that treats rehosting as coordinated emulation task of both target services and their user-space dependencies. By leveraging the firmware's own initialization logic and systematically tracing emulation failures to their underlying causes, FIRMWELL addresses fundamental limitations in existing approaches. Our comprehensive evaluation demonstrates that FIRMWELL successfully rehosts 1.6-8x more services than state-of-the-art tools while reducing analysis time by up to 8.4x. The discovery of 67 zero-day vulnerabilities validates FIRMWELL's practical impact on firmware security analysis at scale.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable comments to improve our paper. This project is supported by National Natural Science Foundation of China (Grant No.62472302 and 92467201).

ETHICS CONSIDERATIONS

During fuzzing, we responsibly reported all confirmed zero-day vulnerabilities to vendors through their recommended channels, ensuring descriptions are accurate, reproducible, and informative. We collaborated with vendors on potential fixes, respecting their timelines to minimize user risk. Adhering to open science policy, we share insights without exposing sensitive information. We comply with relevant laws and regulations, updating our processes to align with industry best practices. These measures help us fulfill our ethical responsibilities and positively impact cybersecurity.

REFERENCES

[1] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[2] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: An analysis of {IoT} devices on home networks," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 1169–1185.

[3] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. 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 2019)*, 2019, pp. 135–150.

[4] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–36, 2021.

[5] Y. Ji, M. Elsabagh, R. Johnson, and A. Stavrou, "{DEFInit}: An analysis of exposed android init routines," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3685–3702.

[6] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, vol. 1, 2016, pp. 1–1.

[7] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Annual computer security applications conference*, 2020, pp. 733–745.

[8] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Bas1que, F. Dong, Z. Smith *et al.*, "Greenhouse: Single-Service rehosting of Linux-Based firmware binaries in User-Space emulation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5791–5808.

[9] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.

[10] A. Costin, A. Zarras, and A. 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*, 2016, pp. 437–448.

[11] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114.

[12] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. 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*, 2022, pp. 417–428.

[13] T. kernel development community, "Kernel support for miscellaneous binary formats (binfmt_misc)," https://docs.kernel.org/admin-guide/binfmt-misc.html, 2024.

[14] QEMU, "Limited support for mips clone syscall in qemu user mode," https://gitlab.com/qemu-project/qemu/-/issues/2112, 2024.

[15] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multi-binary interactions in embedded firmware," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1544–1561.

[16] ReFirmLabs, "Firmware analysis tool," https://github.com/ReFirmLabs/binwalk, 2023.

[17] N. S. Agency, "Firmware Analysis and Comparison Tool," https://github.com/fkie-cad/FACT_core/tree/master/, 2025.

[18] R. Helmke, E. Padilla, and N. Aschenbruck, "Mens Sana In Corpore Sano: Sound Firmware Corpora for Vulnerability Research," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'25)*. San Diego, California, USA: The Internet Society, 2025. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2025-669-paper-1.pdf

[19] Y. Régis-Gianas, N. Jeannerod, and R. Treinen, "Morbig: A static parser for posix shell," in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*, 2018, pp. 29–41.

[20] N. S. Agency, "Ghidra," https://ghidra-sre.org/, 2025.

[21] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, vol. 1, 2015, pp. 1–1.

[22] I. Angelakopoulos, G. Stringhini, and M. Egele, "Pandawan: Quantifying progress in linux-based firmware rehosting," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.

[23] Y. Zheng, Z. Song, Y. Sun, K. Cheng, H. Zhu, and L. Sun, "An efficient greybox fuzzing scheme for linux-based iot programs through binary static analysis," in *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2019, pp. 1–8.

[24] threat9, "Exploitation framework for embedded devices," https://github.com/threat9/routersploit, 2022.

[25] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[26] S. F. Conservancy, "selenium automates browsers. That's it!" https://www.selenium.dev/, 2024.

[27] I. Angelakopoulos, G. Stringhini, and M. Egele, "{FirmSolo}: Enabling dynamic analysis of binary linux-based {IoT} kernel modules," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5021–5038.

[28] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1237–1254.

[29] C. Cao, L. Guan, J. Ming, and P. Liu, "Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation," in *Annual Computer Security Applications Conference*, 2020, pp. 746–759.

[30] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted firmware rehosting for embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 321–338.

[31] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using precise MMIO modeling for effective firmware fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1239–1256.

[32] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic firmware emulation through invalidity-guided knowledge inference," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[33] W. Zhou, L. Zhang, L. Guan, P. Liu, and Y. Zhang, "What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3269–3283.

[34] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware re-hosting through abstraction layer emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1201–1218.

[35] L. Seidel, D. Maier, and M. Muench, "Forming faster firmware fuzzers," in *USENIX Conference on Security Symposium*, 2023.

[36] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti, "PIE: Parser identification in embedded systems," in *Annual Computer Security Applications Conference (ACSAC'15)*, Dec. 2015.

[37] L. Chen, Y. Wang, Q. Cai, Y. Zhan, H. Hu, J. Linghu, Q. Hou, C. Zhang, H. Duan, and Z. Xue, "Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 303–319.

[38] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "{BootStomp}: On the security of bootloaders in mobile devices," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 781–798.

[39] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 463–478.

[40] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 430–441.

[41] K. Cheng, Y. Zheng, T. Liu, L. Guan, P. Liu, H. Li, H. Zhu, K. Ye, and L. Sun, "Detecting vulnerabilities in linux-based embedded firmware with sse-based on-demand alias analysis," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 360–372.

[42] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu, and L. Sun, "Fits: Inferring intermediate taint sources for effective vulnerability analysis of iot device firmware," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023, pp. 138–152.

[43] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A {Large-scale} analysis of the security of embedded firmwares," in *23rd USENIX security symposium (USENIX Security 14)*, 2014, pp. 95–110.

[44] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 480–491.

[45] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Oct. 2017.

[46] H. Xiao, Y. Zhang, M. Shen, C. Lin, C. Zhang, S. Liu, and M. Yang, "Accurate and efficient recurring vulnerability detection for iot firmware," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3317–3331.

[47] A. Qasem, M. Debbabi, and A. Soeanu, "Octopustaint: Advanced data flow analysis for detecting taint-based vulnerabilities in iot/iiot firmware," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2355–2369.

[48] W. Gibbs, A. S. Raj, J. M. Vadayath, H. J. Tay, J. Miller, A. Ajayan, Z. L. Basque, A. Dutcher, F. Dong, X. Maso *et al.*, "Operation mango: Scalable discovery of {Taint-Style} vulnerabilities in binary firmware services," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7123–7139.

[49] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 484–500.

[50] H. Liu, S. Gan, C. Zhang, Z. Gao, H. Zhang, X. Wang, and G. Gao, "Labrador: Response guided directed fuzzing for black-box iot devices," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 127–127.

[51] X. Feng, R. Sun, X. Zhu, M. Xue, S. Wen, D. Liu, S. Nepal, and Y. Xiang, "Snipuzz: Black-box fuzzing of iot firmware via message snippet inference," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 337–350.

[52] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[53] K. Liu, M. Yang, Z. Ling, Y. Zhang, C. Lei, J. Luo, and X. Fu, "Riotfuzzer: Companion app assisted remote fuzzing for detecting vulnerabilities in iot devices," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2341–2354.

[54] X. Ma, L. Luo, and Q. Zeng, "From one thousand pages of specification to unveiling hidden bugs: Large language model assisted fuzzing of matter {IoT} devices," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4783–4800.

[55] OpenWrt, "Init scripts," https://openwrt.org/docs/techref/initscripts, 2024.

Appendix A
Additional Design Details and Experimental Results

A. Firmware Pre-processing

After extracting the firmware, Firmwell identifies two types of programs in the firmware's filesystem to assist with rehosting.

Target Service Identification Firmwell currently focuses on rehosting three main types of network services within Linux-based firmware: HTTP, UPnP, and DNS. And analysts can specify other program names as targets. We compiled a list of network service programs based on previous work [7], [8], [37] and manually analyzed firmware for functions such as `bind` and `recv` to identify a set of executable names for common networked services. We discovered that some firmware includes legacy programs, such as two different HTTP service programs coexisting in the filesystem. Manual inspection revealed that one of these programs is not invoked by any other program. In such cases, we consider all identified programs as target services and exclude the legacy programs from subsequent dynamic analyses (see Section III-B).

Init Binary Identification Since the firmware runs on vendor-customized kernels, the identification of the init binary can be firmware-specific. Firmwell uses a two-step approach to identify the init binary within the firmware. If

TABLE IX: Number of Web Servers Rehosted by FIRMAE, GREENHOUSE, and FIRMWELL on the Greenhouse Dataset. The number of Unpack is from FIRMWELL.

Brand	# of Images	Unpack	FirmAE			Greenhouse			Pandawan			FIRMWELL		
			Execute	Connect	Interact	Execute	Connect	Interact	Execute	Connect	Interact	Execute	Connect	Interact
ASUS	845	812	829	454	24	790	757	755	71	32	27	794	794	**786**
Belkin	63	63	57	23	8	61	47	**34**	24	10	0	38	38	32
D-Link	1,416	1,006	1,074	579	539	826	523	469	249	138	131	856	627	**609**
Linksys	91	70	67	43	35	66	20	19	22	13	3	48	48	**43**
Netgear	2,709	2,235	2,483	1,483	1,217	2,179	1,367	954	482	246	206	2,152	2,061	**2,004**
Tenda	172	153	161	15	9	138	69	**60**	41	0	0	68	48	48
TP-Link	1,047	913	955	536	448	819	188	175	330	167	122	594	594	**593**
TRENDnet	728	466	664	296	248	500	311	235	63	19	16	425	330	**299**
Zyxel	20	19	19	8	6	18	7	6	3	0	0	16	16	**11**
Total	7,098	5,737	6,309	3,437	2,534	5,397	3,289	2,707	1,285	625	505	4,991	4,556	**4,425**

TABLE X: Different Initialization Process Type.

Init Path	File Type	Target Binary	Target Bash
/sbin/preinit	Sym Link	/sbin/rc	-
/*bin/init	Binary	/bin/init or /sbin/init	-
/sbin/init	Sym Link	/bin/busybox	/etc/init.d/rcS, /etc/system/sysinit
/sbin/init	Sym Link	/sbin/rc, /sbin/rcd, /sbin/procd, /sbin/rc_app/rc_apps	-

TABLE XI: The number of Rehosted Web Servers W/O User Space Resource Preparation. Abbreviations refer §II-A.

Brand	w/o EXE	w/o DYN	w/o IPC	Total
ASUS	1,168	336	1,180	1,579
AVM	0	0	0	0
Belkin	24	31	31	32
D-Link	444	454	390	1,116
EDIMAX	5	17	11	19
EnGenius	15	2	53	69
Linksys	80	48	51	136
NETGEAR	672	830	1,084	2,251
Tenda	43	41	36	48
TP-Link	270	491	117	865
TRENDnet	157	222	241	363
Ubiquiti	1	0	1	1
Zyxel	3	10	6	11
Total	**2,882**	**2,482**	**3,201**	6,490

kernel extraction is not feasible, FIRMWELL applies an empirical heuristic based on manual analysis of 14,049 firmware images. In practice, embedded firmware systems invoke a diverse set of init binaries, including at least eight types, such as `/sbin/preinit`, `/sbin/init`, `/bin/init`, `/etc/init`, `/bin/busybox` (as used in early OpenWrt devices [55]), and vendor-specific binaries like `/sbin/rc`. This heuristic enables FIRMWELL to reliably set the emulator's entrypoint to the appropriate init binary across a wide range of firmware images, particularly when kernel extraction is not possible.

B. Experimental Data on Greenhouse Dataset

Table IX presents the HTTP service rehosting results for four tools evaluated on the Greenhouse dataset, which comprises 7,098 Linux-based firmware images spanning nine brands. Among the 5,737 unpacked firmware images, the overall rehosting rates are as follows: FirmAE achieves 44% (2,534), Greenhouse 47% (2,707), Pandawan 9% (505), and FIRMWELL 77% (4,425). Across nine vendors, FIRMWELL outperforms the comparison tools on seven, with slightly lower success rates than GREENHOUSE only for Belkin and Tenda. These results are consistent with the trends in Table II, confirming the generalizability of our findings.

TABLE XII: Details of the zero-day vulnerability discovered by FIRMWELL. BOF=Buffer Overflow (CWE-120), UA=Uncaught Exception (CWE-248), NPD=Null Pointer Dereference (CWE-476), RA=Reachable Assertion (CWE-617).

Brand	Firmware	Binary	CWE	#of Vul
ASUS	FW_RT_AC1200_300438010931	/usr/sbin/httpd	NPD	1
	FW_RT_AC1300UHP_300438252504	/usr/sbin/lighttpd	NPD	1
	FW_RT_AC2400_300438252516	/usr/sbin/httpd	NPD	1
	FW_RT_AC52_300438270638	/usr/sbin/httpd	NPD	1
	FW_RT_AC66W_300438252287	/usr/sbin/lighttpd	NPD	1
	FW_RT_AC85MR_300438252272	/usr/sbin/httpd	NPD	1
	FW_RT_N11P_B1_300438010931	/usr/sbin/httpd	NPD	1
	FW_RT_N16_30043807378	/usr/sbin/lighttpd	BOF	1
	FW_RT_N18U_300438252288	/usr/sbin/lighttpd	NPD	1
	FW_RT_N300_B1_300438010931	/usr/sbin/httpd	NPD	1
	FW_RT_N800HP_300438252242	/usr/sbin/httpd	NPD	1
D-Link	DAP_1520_REVA_FIRMWARE_1.10B04	/sbin/lighttpd	NPD	1
	DIR-860L_REVB_FIRMWARE_2.04.B04	/sbin/httpd	NPD	1
	DCS-932L_REVB_FIRMWARE_2.18.01	/bin/alphapd	BOF	1
	DAP_2555_REVA_FIRMWARE_1.20	/sbin/httpd	BOF	1
	DHP_W306AV_FIRMWARE_1.0.1	/usr/bin/lighttpd	BOF	1
	WBR_1310_REVD_FIRMWARE_4.13	/sbin/httpd	NPD	1
Linksys	FW_RE1000v2_v2.0.04.001_20180227	/bin/lighttpd	NPD/BOF	2
	FW_RE4000W_v1.0.01.001_20180321	/bin/lighttpd	NPD	1
	FW_WES610N_v2.0.01.004_20130719	/bin/lighttpd	NPD	2
NETGEAR	DG834Gv5_V1.6.01.34	/usr/sbin/httpd	NPD	2
	EX6150_V1.0.0.46_1.0.76	/usr/sbin/httpd	BOF	1
	JWNR2000_Firmware_Version_1.0.0.7	/bin/boa	RA	2
	MBR624GU_Firmware_Version_6.01.30.59	/usr/sbin/httpd	RA	1
	R6200_V1.0.1.58_1.0.44	/usr/sbin/httpd	NPD	1
	R6200v2_V1.0.3.12_10.1.11	/usr/sbin/httpd	NPD	1
	R7000P_V1.2.0.22_1.0.78	/usr/sbin/httpd	BOF	1
	RS400_V1.5.1.88_10.0.58	/usr/sbin/httpd	NPD	1
	WGR612_Firmware_Version_1.0.1.2	/bin/boa	UE	2
	WGT624v4_Firmware_Version_2.0.13_2.0.15	/usr/sbin/httpd	RA	3
	WN1000RP_V1.0.0.52	/bin/uhttpd	NPD	1
	WNR2200_V1.0.1.102	/bin/uhttpd	BOF	1
	WNDR3700_V1.0.7.98_WW_	/usr/sbin/uhttpd	NPD	1
	WNDRMAC_Firmware_Version_1.0.0.22	/usr/sbin/uhttpd	BOF	3
	WPN824V3_V1.0.8_1.0.7NA	/bin/boa	UE	2
	XAVN2001_V0.4.0.7	/usr/sbin/uhttpd	NPD	2
TRENDnet	FIRMWARE_TEW_410APBPLUS_1.3.06B	/usr/sbin/httpd	NPD	3
	FW_TEW_638APB_V2_1.2.7_	/bin/goahead	NPD	1
	FW_TEW_818DRU_v1_1.0.14.6_	/usr/sbin/httpd	NPD	1
	FW_TI_G102i_v1_1.0.8.S0_	/usr/sbin/lighttpd	NPD	1
	FW_TI_G642i_v1_1.0.7.S0_	/usr/sbin/lighttpd	NPD	1
	FW_WL500gpv2_3044_TW	/usr/sbin/httpd	NPD	1
	TEW_411BRPplus_2.07	/usr/sbin/httpd	NPD	2
	TEW_637AP_V2_FW1.3.0.106_	/bin/goahead	NPD	2
Tenda	US_F452V1.0BR_V1.0.0.3_en_8097_TD	/bin/httpd	NPD	1
	US_FH1202V1.0BR_V1.2.0.14_408_EN_TD	/bin/httpd	NPD	3
	US_N80_W568Rbr_V1.0.1.17_6610_TDE	/bin/httpd	NPD	1
	US_W15EV1.0br_V15.11.0.5	/bin/httpd	NPD/BOF	3
Total	——	——	——	67

C. Analysis of User-Space Dependencies for Rehosting

We evaluated the impact of different types of dynamically generated user-space dependencies on firmware service rehosting by removing each type individually and rerunning the HTTP service to assess functionality. Table XI displays the number of HTTP services rehosted by FIRMWELL after the removal of specific user-space resources. The results show that dynamically generated files (DYN) are the most critical, as only 2,482 (38%) services remain functional when they are removed. Execution configurations (EXE), which include command line arguments, environment variables, and relative paths, have a moderate impact, with 2,882 (44%) services still operational. Inter-process communication (IPC) dependencies are also important, as 3,201 (49%) services continue to function without them. This analysis highlights the essential role of correctly managing user-space dependencies to achieve successful firmware rehosting.

D. 0-day Vulnerability Details

Table XII shows the 67 zero-day vulnerabilities discovered by FIRMWELL in 48 firmware images. We responsibly disclosed these vulnerabilities to the respective vendors. Among them, 10 vulnerabilities have been confirmed by vendors and assigned CVE IDs.

E. Discussion of Intra-Process Communication for Rehosting

In this paper, we focuses on analyze peer processes that utilize IPC mechanisms that enable rich data exchange in user-space, such as files, sockets, and shared memory. Synchronization-oriented mechanisms, like signals and semaphores, are excluded from our analysis, as they primarily serve to manage notifications and resource control, without directly impacting process functionality. We also omit parent-child IPC mechanisms, including pipes and intra-process communication (*i.e.,* thread communication), which are under the control of the parent process and are typically terminated upon its exit. FIRMWELL's rehosting framework operates at the process level, reasoning and fixing misemulation resources both for parent and child processes. Any missing user-space dependencies related to parent-child IPC mechanisms or intra-process communication are resolved through the correction of system-level misemulations.

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The artifact is available on Zenodo: https://doi.org/10.5281/zenodo.17083186. And the Docker image of FIRMWELL is accessible via:

```
$ docker pull ghcr.io/qc9c/firmwell:ae
```

2) Hardware dependencies: Minimum requirements for FIRMWELL are 1 CPU core, 4 GiB memory, and 100 GiB disk space per rehosting task. For large-scale evaluation, a Kubernetes cluster with at least 1,000 CPU cores was used (each pod: 1 core, 4 GiB RAM). The firmware dataset requires 428 GiB of storage, and 1 TiB of additional disk space is recommended for analysis outputs.

3) Software dependencies: For local evaluation, FIRMWELL only requires Docker, as we provide a pre-built Docker image of FIRMWELL and all necessary third-party dependencies. For batch or parallel analysis, a Kubernetes cluster with shared storage is required. All experiments were validated on Ubuntu 20.04.

4) Benchmarks: FIRMWELL was run on a dataset of 14,049 firmware images crawled from 13 different vendors. This dataset is hosted privately as part of the artifact, contact the authors if access is needed.

B. Artifact Installation & Configuration

1) Prerequisites: FIRMWELL has been packaged as a Docker container. To deploy FIRMWELL, the following software must be installed on the host machine:

- *Docker (tested on version 20.10.21, build 20.10.21)*
- *Docker Compose (tested on version 1.29.2, build 5be-cea4c)*

2) Installation: Load the Docker image.

```
$ docker load -i firmwell.tar
```

Start the container in privileged mode.

```
$ docker run --privileged -v /dev:/host/dev
-it firmwell:ae bash
```

Inside the Docker container, initialize the analysis environment by executing the setup script:

```
$ bash /fw/docker_init.sh
```

Additionally, reference images for state-of-the-art tools (`firmae`, `pandawan`, `greenhouse`) are available via:

```
$ docker pull ghcr:<sota>:latest
```

C. Experiment Workflow

At a high-level, FIRMWELL rehosts each input firmware image and evaluates the result using the Rehosting Checker (see Section III-A). Firmware images that pass this check are further analyzed for vulnerabilities.

D. Major Claims

- **(C1): Rehosting Performance** FIRMWELL is capable of successfully rehosting 68% of HTTP services from firmware images that can be unpacked from the dataset. This result is supported by the experimental evaluation E1, as presented in Table II (RQ1, Section V-A) of our paper.
- **(C2): Component Effectiveness** The key design modules of FIRMWELL each contribute significant and complementary effectiveness. Disabling any major module results in a marked reduction in the overall success rate, with the ablated success rate ranging from 91% down to 34% of the original result, depending on the specific component removed. This result is supported by the experimental evaluation E2, as presented in Table IV (RQ2, Section V-B) of our paper.

TABLE XIII: Ablation Experiment Arguments (correspond to Table IV in the paper).

Arguments	Description
`--baseline`	Enable only the Section III-A module
`--wo_32`	Disable the Section III-B module
`--wo_33`	Disable the Section III-C module
`--wo_create`	Disable the "CREATE" strategy in Section III-C
`--wo_infer`	Disable the "INFER" strategy in Section III-C
`--wo_reuse`	Disable the "REUSE" strategy in Section III-C
`--wo_peer`	Disable the "FIX-IN-PEER" strategy in Section III-C

- **(C3): Real-World Application** The firmware images rehosted by FIRMWELL enable dynamic analysis for real-world vulnerability discovery. FIRMWELL identified 1,335 N-day vulnerabilities with RouterSploit and 67 zero-day vulnerabilities via AFL++ fuzzing. This result is supported by the experimental evaluation E3 and E4, as presented in RQ3 (Section V-C) of our paper.

E. Evaluation

1) Experiment (E1): **[Firmware Rehosting] [10 human-minutes + 1 compute-hour]**: Evaluating FIRMWELL's ability to rehost an input firmware image and assess the functionality of its HTTP service using the Rehosting Checker.

[Preparation] Launch the Docker container as described in the Artifact Installation section, and then copy the firmware image to be analyzed into the container.

[Execution] Execute the following command to rehost a target firmware image using FIRMWELL.

```
$ /fw/run.sh <brand> <image-path>
```

[Results] FIRMWELL reports the rehosting status in the console. A message of the form `REHOST STATUS - <sha256> - SUCCESS` indicates that the HTTP service has been successfully rehosted and verified. The corresponding rehosted image is stored under `/tmp/results/<sha256>` in the container.

2) Experiment (E2): **[Ablation Study] [10 human-minutes + 1 compute-hour]**: Evaluation of component-wise effectiveness via module ablation.

[Preparation] Launch the Docker container as described in the Artifact Installation section, and then copy the firmware image to be analyzed into the container.

[Execution] To conduct ablation experiments corresponding to Table IV of the paper, specify the appropriate arguments in the `args` field of the configuration file for each experimental setting. Each argument (e.g., `--baseline`) enables or disables specific FIRMWELL modules as summarized below Table XIII.

```
$ /fw/run.sh <brand> <image-path> <args>
```

[Results] FIRMWELL reports the rehosting status in the console. A message of the form `REHOST STATUS - <sha256> - SUCCESS` indicates that the HTTP service has been successfully rehosted and verified.

3) Experiment (E3): **[N-day Vulnerability Detection] [10 human-minutes + 1 compute-hour]**: Automated detection of known (N-day) vulnerabilities in rehosted firmware.

[Preparation] Launch the Docker container as described in the Artifact Installation section, and then copy the firmware image to be analyzed into the container.

[Execution] Execute the following command to rehost a target firmware image and perform N-day vulnerability detection using FIRMWELL.

```
$ /fw/run.sh <brand> <image-path> --rsf
```

[Results] After the analysis completes, a summary of all detected N-day vulnerabilities is saved in `/tmp/processed_data/vulnerable.csv`, corresponding to the results in Section IV-C and Table VII.

4) Experiment (E4): **[Fuzz-testing] [20 human-minutes + 24 compute-hour]**: Automated fuzz-testing of rehosted firmware images for zero-day vulnerability discovery.

[Preparation] After completing Experiment (E1), FIRMWELL exports the rehosted image to `/tmp/results/<sha256>`.

Set up the fuzzing environment and build a dedicated Docker image using the following command:

```
$ python /root/build_fuzz_img.py -name fuzz
  -f /results/<sha256>.tar.gz
```

[Execution] Launch the fuzz-testing process by starting the generated Docker image:

```
$ docker run --ulimit nofile=2048:2048
  --cap-add NET_ADMIN --cap-add SYS_ADMIN
  --security-opt seccomp=unconfined
  -v /scratch:/scratch -i fuzz /fuzz.sh
```

[Results] The fuzzing workflow will begin and AFL++ will output real-time fuzzing status. Note that not all rehosted images are directly amenable to fuzzing, since certain firmware may require additional engineering or customization of AFL++ to ensure proper operation.