Target-Centric Firmware Rehosting with Penguin

Andrew Fasano[†§], Zachary Estrada[†✉], Luke Craig[†], Ben Levy[†], Jordan McLeod[†], Jacques Becker[†], Caden Kline[†], Elysia Witham[†], Cole DiLorenzo[†], Ali Bobi[†], Dinko Dermendzhiev[||], Tim Leek[†], Wil Robertson[§]
MIT Lincoln Laboratory[†] Northeastern University[§] Georgia Institute of Technology[||]
{fasano, zje, luke.craig}@mit.edu Benjamin.Levy@ll.mit.edu jmcleod@mit.edu
{Jacques.Becker, Caden.Kline, Elysia.Witham, Cole.DiLorenzo, Ali.Bobi}@ll.mit.edu
dermendzhiev@gatech.edu wkr@ccs.neu.edu tleek@ll.mit.edu

***Abstract*—Firmware rehosting allows firmware to be executed and dynamically analyzed. Prior rehosting work has taken a "one-size-fits-all" approach, where expert knowledge is baked into a tool and then applied to all input firmware. Penguin takes a new, target-centric approach, building a whole-system rehosting environment tailored to the specific firmware being analyzed. A rehosting environment is specified by a configuration file that represents a series of transformations applied to the emulation environment. The initial rehosting configuration is derived automatically from analyzing the filesystem of an extracted firmware image, providing target-specific values such as directories, pseudofiles, and NVRAM keys. This approach allows Penguin to rehost systems from a wide variety of vendors. In tests on 13,649 embedded Linux firmware images from 69 different vendors and 8 architectures, Penguin was able to build rehosting environments that work for 75% more firmware than the prior state of the art. We implement a configuration minimizer that finds required transformations and show that most firmware require only a small number of transformations, with variation across vendors.**

I. Introduction

Embedded systems are present throughout daily life. Their prevalence has led to them being an important part of individuals' security, handling sensitive information from banking credentials to medical data. Being able to perform rigorous security assessments of those devices is both a monumental and critical task necessary to reduce the security risk those systems pose to users. However, utilizing physical copies of devices for security analysis can be costly, is not scalable, and cannot provide sufficient detail for a deep assessment of a device.

Dynamic analysis of a system's software can reveal a host of security issues, but vendors rarely provide access to an appropriate emulator for this purpose. This means that, in order to perform dynamic analysis, an analyst must first build a bespoke emulator, a daunting task requiring expert knowledge at nearly every step, increasing both the time and cost of analysis dramatically.

The costs of producing an emulation environment suitable for analysis have resulted in a plethora of research into the subject of fully or partially automating emulation. This process, known as rehosting, can be simplified using tooling that leverages similarities between embedded systems in order to derive additional information about the requirements a given firmware imposes on a virtual environment. Linux embedded systems, in particular, have been the subject of much research due to both being so prevalent and having multiple stable interfaces to build against, but also due to their use of open source components.

The process of rehosting can be viewed as a series of transformations applied to a filesystem and runtime environment. The goal of rehosting then is to find the transformations that produce a system suitable for dynamic analysis. In the past, researchers would analyze a set of firmware, craft a set of specific transformations that worked for those systems, and then apply those transformations to other systems hoping they would work. This approach is ineffective given the diverse firmware landscape. E.g., an analyst may wish to rehost firmware that is dissimilar to firmware they previously rehosted (i.e., from a new vendor).

Rather than providing analysts with "one-size-fits-all" transformations applied to all firmware, we see benefit in providing granular controls that apply individualized changes to the analysis environment. Additionally, it is difficult to understand system behavior due to the quantity of information that must be processed to find rehosting failures. Thus, feedback throughout the rehosting process is crucial to help guide the analyst.

We introduce Penguin, a new approach to rehosting embedded Linux systems where rehosting transformations are made explicit, are tailored to the firmware in hand, and can be easily modified or shared by users. We pair this approach with a suite of dynamic analysis plugins designed to aid in identifying failures in a rehosting.

The design of Penguin is guided by the following goals:

1) **Configuration not code**: The transformations for a rehosting should be specified in a configuration that can be understood and modified by both humans and machines, removing the need for the user to customize kernel or

✉ Corresponding author. Email: zje@mit.edu

Workshop on Binary Analysis Research (BAR) 2025
28 February 2025, San Diego, CA, USA
ISBN 979-8-9919276-4-2
https://dx.doi.org/10.14722/bar.2025.23xxx
www.ndss-symposium.org

emulator code. The virtual environment should be fully represented by the configuration to enable reproducible and easily distributable rehostings.

2) **Enable dynamic analysis**: The platform should offer a standard interface for developing and deploying dynamic analyses to learn about the target's behavior during rehosting and subsequent analysis.
3) **Unprivileged, scalable execution**: The platform should not require root access or privileged containers. This is necessary to function in a wide variety of environments, including high-performance computing clusters for large-scale experiments. Existing rehosting tools require running in a privileged context for firmware repackaging and networking.
4) **Vendor-Agnostic Intervention Design**: While the particular values and applications of rehosting transformations are often vendor-specific, their high-level concepts are often universal. We strive to make our approach sufficiently generic to work across vendors, deriving values from analysis rather than hardcoding expert knowledge when possible.

II. Background and Related Work

A. Firmware Rehosting

A variety of approaches have been taken to provide solutions to the rehosting problem, each offering different tradeoffs. For example, many firmwares can be viewed as standard Linux systems in which only userspace programs are of interest to security analysts. Consequently, rehosting is reduced to analyzing individual Linux binaries and addressing missing environmental dependencies, as demonstrated by Greenhouse [1]. Other approaches seek to maintain the ability to perform whole-system analysis without incurring the burden of emulating arbitrary Linux kernels by swapping the firmware-provided kernel for one provided by the virtual environment, such as in Firmadyne [2] and its successor FirmAE [3]. Other work, such as FirmSolo, attempts to make kernel modules runnable by building a compatible kernel [4]. Whole-system analysis comes with a performance penalty, however is necessary to enable the discovery of vulnerabilities that only arise across multiple layers and interactions within the system [5].

Generalized approaches to rehosting need some form of interface to build around, however not all solutions choose the interfaces provided by an Operating System (often Linux). Others choose Memory-Mapped Input/Output (MMIO) or similar means of interfacing at the hardware layer, such as PRETENDER [6] and P^2IM [7]. Modeling at this layer allows a virtual environment to be operating system agnostic in exchange for reduced ability to build an understanding of introspected data.

When the goal of rehosting is dynamic analysis of a system's software stack we have found it more fruitful to abstract away missing functionality with OS interactions rather than building precise device models. Therefore, for Penguin we have chosen an approach in line with Firmadyne/FirmAE: whole-system emulation with kernel replacement.

B. Transferability of Rehosting Interventions

"One-size-fits-all" rehosting approaches apply a set of transformations to all firmwares rehosted by the system, possibly with some hard-coded decision making. These approaches can be successful in generating broad results across a large corpus. What if an analyst, however, is interested in a particular device that is not well-represented by the devices the one-size-fits-all system was designed for? Identifying and remediating rehosting failures in one-size-fits-all systems can be extremely time-consuming and expensive [8]. We investigate the transferability of rehosting interventions to answer the question: *do one-size-fits-all approaches solve the rehosting problem for all vendors, or just the ones they are developed against?*

Firmadyne [2] and its successor, FirmAE [3], both conduct large-scale evaluations of thousands of router firmware images to evaluate their approaches. Firmadyne examines firmware from 42 vendors, but over 93% of their labeled successes come from four vendors: D-Link, Netgear, TP-Link, and TRENDnet. FirmAE is evaluated on routers and IP cameras using firmware from these same four vendors alongside four additional vendors: ASUS, Belkin, Linksys, and Zyxel.

While FirmAE successfully starts webservers in 79% of their tests, we find the framework only succeeds in 33% of tests on a more diverse dataset (Section V-A). If we exclude the eight aforementioned vendors from the diverse dataset, FirmAE's success rate drops to just under 9%, whereas Penguin succeeds on 52% (Section V). This implies FirmAE primarily introduces vendor-specific interventions, rather than generalized solutions to rehosting Linux firmwares. While the tools and techniques needed to resolve rehosting failures can be common across vendors, the actual specifics of how to address failures are not, which is why we believe a target-centric approach is necessary.

III. System Design

Penguin generates a configuration file that specifies the details of a rehosting environment. The initial configuration combines established defaults with information learned from static analysis of the target filesystem. A configuration is then *realized* into a rehosting environment and filesystem. This rehosting is executed and dynamic analysis results are recorded for the user. Through observation of the rehosting's runtime behavior, failures can be identified and the configuration file can be refined to address those failures. Using this iterative rehosting process, Penguin aims to be a framework for tailoring a rehosting to a specific firmware. To reproduce or share a rehosting a user simply needs the configuration file and original firmware image. The stages of the Penguin rehosting framework are shown in Fig. 1.

A. Filesystem Analysis

We perform a series of static analyses to identify the firmware's filesystem and derive an initial rehosting configuration. Once a filesystem is generated, it is necessary to identify

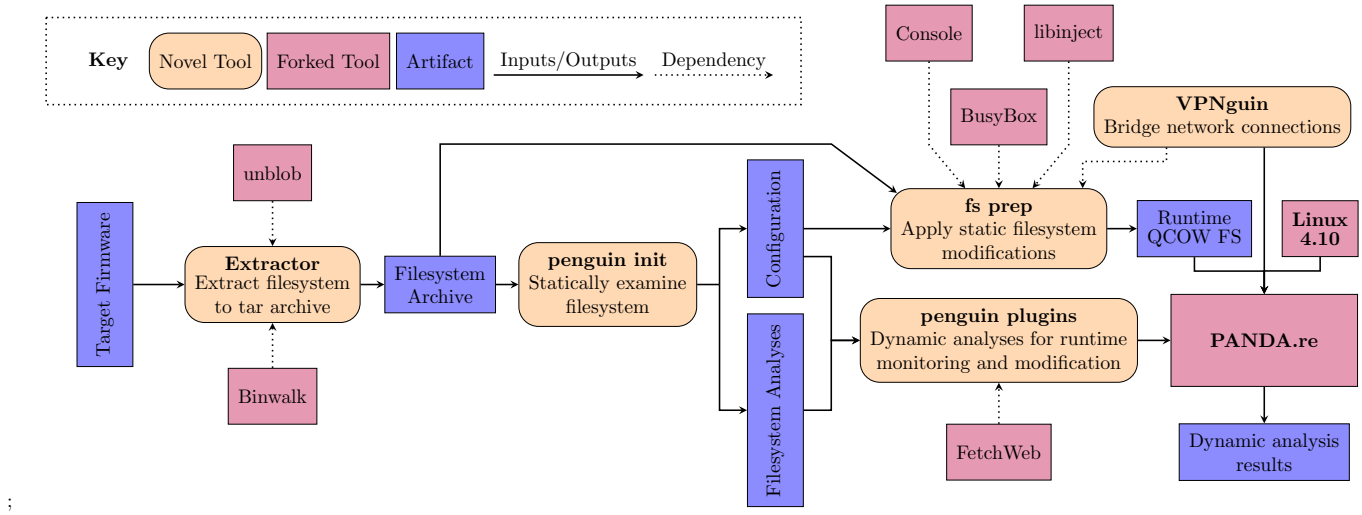

Fig. 1: Rehosting a firmware with Penguin.

the original system's architecture, bit-width, and endianness to select appropriate parameters for CPU emulation. Additionally, the *init* program is identified.

Penguin extends the static analyses from both Firmadyne and FirmAE as well as supports several new analyses. While some analyses from Firmadyne and FirmAE are tailored to specific vendors (or even firmware), our novel analyses are designed to be firmware agnostic. The (configurable) suite of filesystem analyses in Penguin are summarized in Table I and given more detailed treatment in Appendix C.

B. Rehosting as a Configuration

After the filesystem analyses are complete, the Penguin user has an initial rehosting configuration. A Penguin rehosting is a YAML file that is paired with a repackaged filesystem to precisely specify that rehosting. The configuration capabilities can be broadly categorized into three groups: static filesystem modifications, runtime environment, and dynamic analyses. A brief example of a configuration is in Appendix B

1) Static Filesystem Modifications: Users can define a set of modifications to make to the filesystem before the rehosting starts. Files can be added, deleted, or moved.

2) Runtime Environment: The configuration specifies the emulated environment that the rehosting will execute under. This includes emulator configuration (e.g., which CPU architecture), Linux boot arguments, network device names, and initial NVRAM values. Additionally, the configuration can specify new pseudofiles in */proc*, */dev*, and */sys* as well as models for how those pseudofiles should behave.

3) Dynamic Analyses: While a rehosting runs it can be analyzed to identify failures and inform future refinement to the rehosting configuration. Beyond the rehosting process, dynamic analyses of a rehosted system can also be leveraged to reverse engineer system behavior, identify vulnerabilities, or otherwise exercise the functionality of a system.

IV. Implementation

Penguin's implementation enables configuration-based rehosting, dynamic analysis of target systems, and collection of health metrics. Throughout the entire Penguin process, no special permissions are required and as such we ran the experiments in Section V as an unprivileged user on a computing cluster.

Penguin can rehost Linux-based systems across a variety of architectures (x86, ARM and MIPS), ABIs (e.g., hard and soft float on ARM), bit-widths (32-bit and 64-bit), and byte orders (big and little endian). The implementation could be extended to support other architectures, though many aspects of our approach depend either directly or indirectly on the stable nature of the Linux syscall ABI.

A. Extractor

The rehosting process begins with a binary blob of data representing the system's software. This blob may be drawn from a variety of sources ranging from vendor-provided files (which may be partial updates or entire firmware images) to memory dumps from physical devices. First, a filesystem is extracted from this image, repacked into a usable format, and modified if necessary. Accurate extraction is vital as missing files, incorrect permissions/owners, and broken symlinks may cause execution failure or crashing of critical services.

Binwalk and unblob are two open-source filesystem extractors that recursively scan binary blobs for filesystems and extract their contents. Binwalk [9] has been widely used in the literature for both static [10] and dynamic analysis [2], [3], [11], [1] of firmware images, but its development stalled since 2021, only later to be rewritten. The rewrite (Binwalk v3), while a positive development, was yet to meet feature parity at the time of writing while many of the previous version's extractors suffer from known flaws. Recently, unblob [12] was released as an alternative filesystem extractor designed to improve upon the capabilities of Binwalk. In limited testing,

Analysis Name	Description	Based on	Produces
`Architecture`	Identify the target architecture.	Firmadyne	Single value
`Init`	Identify potential init binaries.	Firmadyne	List of values
`NVRAM`	Identify default NVRAM values.	FirmAE	List of values
`ForceWWW`	Identify command(s) to force start web server(s)	FirmAE	List of values
`Missing Files/Directories`	Create expected directories and files	Novel / FirmAE	List of values
`Shims`	Replace select userspace applications with alternatives.	Novel / Firmadyne	List of values
`Pseudofiles`	Identify references to pseudofiles in */dev*, */sys*, and */proc/*.	Novel	List of values
`ABI`	Identify the target application binary interface (e.g., *armhf*).	Novel	Single value
`Library Functions`	Identify all functions exported by libraries.	Novel	List of values
`Network Interfaces`	Identify possible network interface names.	Novel / Firmadyne	List of values

TABLE I: Static Filesystem Analyses Performed by Penguin.

we found unblob to outperform Binwalk 2.x in terms of both speed and accuracy, often by a wide margin. However, unblob does not support every filesystem format that Binwalk does, and neither tool is designed to preserve filesystem permissions or symlinks.

To work around symlink and permission issues, prior work has taken heavy-handed approaches to fix permissions (e.g., recursively adding execute permissions) [3]. While modifying permissions in that way can be effective, it may also introduce subtle inconsistencies that affect the rehosting process or lead to incorrect analysis results down the line (e.g., false positives on bugs). Even when the filesystem is extracted correctly, the permissions of the user running the extraction tool can affect the permissions of the extracted files.

In order to resolve such issues our extractor runs Binwalk and unblob under *fakeroot* to ensure that permissions are preserved. Within the fakeroot environment, we analyze the extracted filesystems to identify the largest filesystem that appears to be Linux-based [2]. We further check for executable files, a requirement for runnable Linux filesystems. Finally, if both extraction tools produce a filesystem, we take the output from unblob. If no filesystem is found, we report a failure. While still operating under fakeroot we repackage the resulting filesystem into an archive to preserve the permissions between extraction and the subsequent static analysis step.

B. Static Analysis and Configuration Generation

1) Initialization: Penguin analyzes the repackaged filesystem to generate an initial configuration. First, the architecture of the filesystem is identified by examining the architecture of binaries within standard binary directories. If the architecture cannot be identified or if the architecture is not supported, the process halts. Next, a configuration skeleton is generated that specifies the architecture, the OS kernel, and a set of fixed filesystem modifications to add the Penguin in-guest utilities (Section IV-D) to the guest filesystem. A set of default plugins is then added to the configuration.

After those initial steps, the static analysis begins in earnest. The filesystem archive is examined to identify if a set of standard directories are present. Any missing directories are added to the configuration. This stage of the analysis largely implements the filesystem preparation strategy of Chen et al. [2] and Kim et al. [3], but the modifications are written into the user-facing configuration file instead of directly into the filesystem.

Next the guest filesystem is searched for a set of 8 *shim* targets: standard programs that Penguin supports alternative implementations for. These alternatives provide instrumentation, optimize performance, or disable unwanted behavior.

After shimming, the guest filesystem is searched to identify potential *init* binaries and scripts. This search identifies executable files with names containing the string `init` or `start`. While this search is not exhaustive, it is more versatile than prior approaches of simply checking for a hardcoded list of paths [2].

A static analysis then examines each shared object within the guest filesystem using pyelftools [13] to identify exported symbols and identify default NVRAM values exported by libraries. The guest filesystem is also searched for default NVRAM values by identifying NVRAM configuration files. If multiple sources of default NVRAM values are identified, libraries take precedent over the filesystem.

Finally, the static analysis implements FirmAE's [3] technique of statically identifying webservers within the guest filesystem so they can be directly executed. The configuration is extended to create a new script with commands to launch each such service if the `ForceWWW` feature is enabled by the user.

2) File System Preparation: A Penguin configuration file may specify static modifications to the filesystem. The modifications may include adding, removing, or moving files, directories, and symlinks. New scripts or text files may have their contents specified inline in the configuration, or binaries may be copied in from a host file at a specified path. The original filesystem tar is left unmodified throughout the rehosting process, and a QCOW image is generated from the tar using `genext2fs`[1] and `qemu-img`.

C. Penguin Runtime

Given a configuration and an archive of a filesystem to rehost, Penguin will then realize the configuration into rehosting artifacts. This begins by transforming the filesystem as described above. By default, a suite of pre-built Penguin utilities of the appropriate architecture are copied into the guest to enable shell access (console), dynamically intercept library function calls (libinject), instrument shell scripts (busybox), and enable network communication (VPNguin). After the configuration is applied to the filesystem, a virtual disk image

[1] https://github.com/bestouff/genext2fs

specific to that configuration is produced (images are cached per configuration).

After the filesystem is generated, the runtime environment is loaded. The PANDA.re [14] dynamic analysis platform is used for emulation with rehosting enhancements as PyPANDA python plugins [15]. Dynamic analysis plugins are loaded and provided with the configuration. Those plugins are tasked with monitoring the behavior of the guest as it runs and enforcing the configuration. They may coordinate with in-guest logic to dynamically collect the desired information and enforce the desired modifications. For example, the *pseudofiles* plugin coordinates with the guest kernel to track accesses to files in `/dev`, `/proc`, and `/sys`, identify accesses to missing devices, and to model the behavior of devices specified in the configuration. Other plugins offer network vulnerability scanning (Nmap [16]) and web server interaction.

D. Guest Coordination and In-Guest Utilities

During execution, Penguin user space utilities and the kernel directly provide information to Penguin plugins through hypercalls [17]. Though we are executing with PANDA which offers virtual machine introspection (VMI) features, hypercalls provide a reliable mechanism for plugins to passively track and actively modify guest behavior. As our approach to rehosting already requires a custom kernel and additional user space applications, directly modifying those components to provide the desired information is more straightforward than VMI. In more adverserial usecases (e.g., malware analysis) this reduction in fidelity can be unacceptable. Modified files and additional directories are observable effects, even if they are unlikely to impede the operation of a properly-behaving device. We find this tradeoff to be favorable in the vast majority of cases.

1) Instrumented Kernel: Our instrumented kernel extends Linux 4.10 with a custom driver for dynamic pseudofiles and a set of modifications to collect and report internal state to the Penguin runtime plugins. Our *Pseudofiles* driver supports creating pseudofiles in `/dev`, `/proc`, and `/sys` at runtime, using hypercalls. Beyond creating files, the driver also forwards request for reads, writes, and ioctl system calls to the Penguin Pseudofiles plugin so that system call responses can be dynamically crafted. Our kernel also extends the standard Linux memory management logic to report the details of every memory virtual memory area including the name of a backing file and the base address of the area (for measuring coverage).

2) Network Communication: Embedded systems often have hard-coded expectations for network interfaces that are configured during their boot process. For example, a router may assign a static IP address to one interface (i.e., `eth0` for a LAN) and then configure a DHCP server on another interface (i.e., `eth1` for a WAN). Traditional approaches to rehosting [2], [3] have attempted to learn network configurations and reconfigure the emulator and host network stack to allow communication with the rehosted firmware. This approach requires careful analysis of individual firmwares and precise configuration of the host network stack, which requires root privileges and is error-prone. Furthermore, once the network is configured, services of interest may be unreachable due to firewall rules affecting the communication interfaces.

Firmadyne [2] fails to learn the networking configuration of nearly 70% of the firmware images evaluated, prohibiting subsequent analysis. FirmAE [3] identifies a suite of "network arbitrations" to reduce this failure rate including reconfiguring guest firewall rules to allow for analyst connections.

We take a new approach to this problem by configuring the guest with a suite of "dummy" interfaces[2] specified in the rehosting configuration. These dummy interfaces appear to the guest as real network interfaces, but they are not connected to any emulated hardware; they are effectively loopback interfaces. While this approach enables the guest to reconfigure and bind to interfaces, it does not solve the problem of establishing communication with services listening in the emulated guest.

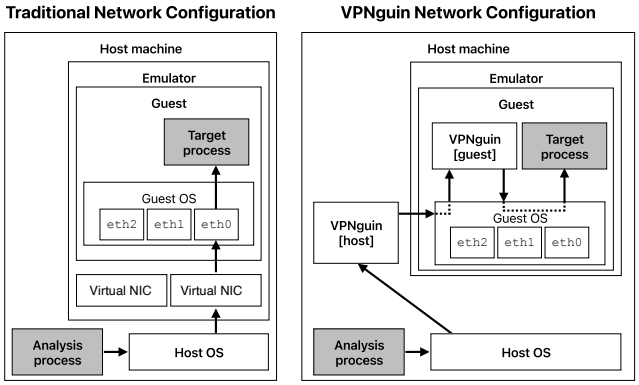

Fig. 2: VPNguin architecture versus traditional rehosting network communication.

We address this problem with *VPNguin*, a novel utility that runs in both the guest and host to bridge network traffic between the two, bypassing the need for emulated network hardware. The two sides of VPNguin communicate over the VirtIO Vsock protocol, a communication channel that bypasses the traditional emulated network stack, and a member of the `AF_VSOCK` family [18], [19]. This approach ensures that the two sides of VPNguin can communicate, regardless of the guest and host network configuration. No special privileges are required by this approach so long as the kernel is compatibile. The architecture of VPNguin is shown next to a traditional rehosting network configuration in Fig. 2.

When paired with the guest's dummy network interfaces, VPNguin enables host-based analyses such as network vulnerability scanners to communicate with guest services listening on any guest interface. Penguin offers an *Nmap* plugin to automate analysis of network-listening guest services.

When the host instance is configured (guest IP address, port, and network protocol), it binds a host socket and launches a thread to bridge traffic for that service. Whenever traffic

[2] https://tldp.org/LDP/nag/node72.html

is received on that socket, VPNguin forwards the traffic and destination metadata to the guest instance over an AF_VSOCK socket. The guest instance sends the traffic to the target service through guest's local network and forwards the response back to the host instance. The host instance sends the guest's response back to the original client that initiated the request.

When guest services bind to multiple IP addresses or ports, VPNguin can be configured to bridge traffic for each. For example, if a firmware configures `eth0` with IP address 192.168.1.1 and `eth1` with 10.0.0.1 and then spawns a network service that listens on both interfaces, VPNguin allows a user to connect to the service on either IP address from the host. This is particularly valuable when services listen on select interfaces or expose different functionality depending on the IP address used (e.g., WAN vs. LAN interfaces).

Unlike the TAP/TUN networking often used in prior work [2], [3], VPNguin does not require elevated host privileges. If the host kernel does not support `AF_VSOCK` or the user does not have permissions to write to `/dev/vsock`, a userspace daemon is used.[3]

3) LibInject: LibInject is a shared library that is injected into the *init* process via `LD_PRELOAD`. The library is significantly based on the LibNVRAM library from Firmadyne [2] and extended by FirmAE [3], but we have rewritten it to work with arbitrary library functions, improve performance and allow for runtime tracking of behavior with hypercalls. LibInject is driven by the Penguin configuration and a user can custom define function responses and even C functions from the Penguin configuration file.

4) Console: Penguin can provide an interactive root shell for the user to interact with the guest. We extend Firmadyne's [2] console to support interactive job control (i.e., `ctrl + c` and `ctrl + z`). This console is configured so that the user's `PATH` prioritizes Penguin's BusyBox utilties over the firmware's.

5) BusyBox: We place a custom BusyBox [20] binary into guest filesystems for two purposes. The first is to ensure a standard suite of commands will be available to a user of the Penguin. The second is to customize the BusyBox shell (*ash*) to perform various dynamic analyses to be reported via hypercalls.

By default, Penguin configurations replace existing `/bin/sh` binaries with a symlink to our instrumented BusyBox. Our BusyBox instrumentation provides script-level code coverage as well as information about the comparisons made during script execution. Comparisons tracked include expected file properties and environment variable values. This functionality is made possible by the lazy expansion of variables. Rather than eagerly expand environment variables utilized in strings and command arguments, BusyBox's *ash* shell embeds the names of variables to be expanded via escape sequences in strings. Those escape sequences are parsed in order to understand which environment variable(s) affect a given conditional and an analyst can see what value is needed to change control flow.

[3]https://github.com/rust-vmm/vhost-device/

E. Penguin Runtime Plugins

Penguin's suite of runtime plugins are designed to monitor and manage the guest environment during emulation. These plugins enforce runtime modifications necessary to ensure the environment acts as specified in the configuration and to collect data for subsequent analysis. These are implemented as PANDA plugins in both Python and C++. A brief description of each plugin is provided below.

The **Core** plugin analyzes the output of the guest system console to identify if a kernel panic has occurred. If a panic is detected, the plugin reports the failure and terminates the rehosting. The plugin also enforces a timeout if one is specified in the configuration file. The plugin manages the processing and dispatching of guest hypercalls to the appropriate plugins.

The **Env** plugin analyzes how guest processes interact with environment variables. The plugin coordinates with library hooks in LibInject, to track whenever environment variables are accessed through the `getenv` function or whenever the contents of `/proc/cmdline` (which contains the boot arguments and can specify environment variables) are compared to another string.

The **Interfaces** plugin tracks the network interfaces a guest attempts to interact with. If a network interface is missing, the plugin proposes adding the missing interface to the rehosting configuration.

The **Mount** plugin tracks failed mount attempts within the guest. Filesystem mounts may fail due to missing storage devices or a lack of kernel support for mounting in-guest filesystems. The plugin optionally makes all `mount` attempts return as if they had succeeded.

The **Nvram** plugin coordinates with the in-guest LibInject to track how values are loaded, stored, and cleared from the guest's NVRAM.

The **Pseudofiles** plugin coordinates with the modified kernel to track accesses to missing pseudofiles in `/dev`, `/proc`, and `/sys` and to model those files as specified in the configuration. If a guest process attempts to access a missing pseudofile, this failure is recorded.

The **NetBinds** plugin analyzes the syscalls issued by the guest to identify whenever a network bind occurs. When such an event is detected, it informs other plugins of the event.

The **VPN** plugin manages the execution and dynamic configuration of the host VPNguin instance. This plugin consumes events from the NetBinds plugin. The VPN plugin can inform other plugins of the newly-available service (e.g., the FetchWeb plugins can be notified that an http server is ready for communication).

The **FetchWeb** plugin can query a webserver using `wget` whenever a TCP bind to port 80 or 443 occurs. The response can be logged for analysis.

The **FICD** plugin implements the Firmware Initialization Completion Detection metric from Pandawan [11], which uses

edit distance of newly spawned processes to determine when system intialization has finished.

In addition to the plugins described above, we have written plugins for collecting coverage (including lines of shell script), health metrics (e.g., number of processes executed), blocking signals, and running nmap. Those plugins are not a part of our evaluation so descriptions have been removed for space considerations.

While trying to build a high-quality rehosting configuration, it may be advantageous to evaluate known-incorrect configurations that are designed to reveal unknown information about the guest. These configurations are *learning analyses*.

One example of a learning analysis is **Dynamic String Search**, designed to recover potential values for strings. When a key (e.g., an environment variable) is identified as potentially interesting, but its value is unknown, this analysis can be used to learn potential values for that key. The user sets that key to a "magic" value. At runtime, dynamic string search analyzes every guest function call and checks if the arguments contain an address pointing to that magic value. If a match is found and the other argument also contains an address pointing to a sequence of null-terminated ascii characters, the function is assumed to be some form of string comparison and that other argument is recorded as a possible value for the key.

F. Configuration Minimization

Penguin's analyses produce a large number of possible configuration options[4] and we anticipate most of those are not actually used in a rehosting. Some configuration options might be superfluous and/or unrelated to the components of interest for security analysis. An analyst might be curious, then, about which configuration options are actually necessary for a system once it has been rehosted.

With firmware executions taking on the order of minutes, it becomes intractable to minimize a large configuration space down to each individual option. For computational efficiency and for user readability, we separate our configurations into individual *patches*. A configuration file can then hierarchically include a number of patches. For example, each analysis in Table I produces a configuration patch as its output. Anecdotally, we have found grouping rehosting transformations into patches useful for separating out manual changes to the configuration.

We minimize configurations by running the firmware with all options in that configuration enabled except for a given patch. *We consider a patch essential if the firmware no longer rehosts without that configuration patch.* The user of the configuration minimizer specifies what the criteria is for a successful rehosting (e.g., a non-empty HTTP response, a certain process executes, etc...). We test all patches in a given configuration this way and our minimizer ensures that patches are orthogonal by splitting overlapping options into distinct patches if necessary. Note that as a performance optimization, this minimizer does assume orthogonality of features, which we do not provide formal guarantees around.

During our experiments, we observed false positives (non-essential features that were labelled as required) due to non-determinism in system execution when sometimes services would fail to start. Based on our observations, we hypothesize that this non-determinism comes from a variety of sources:

- **Process scheduling** Startup often launches a number of processes that all want to do work and then the scheduler non deterministically decides which gets to run
- **Timeouts** System runtime is variable and the timeout during an experiment may be close to that timeout.
- **Memory Accesses** Garbage data is sometimes valid pointers but sometimes is not. Invalid pointers will cause processes to crash.

We increase confidence in our minimization results by attempting to run a firmware multiple times before marking a patch as essential. If a firmware fails to rehost without a given configuration patch ten times in a row we mark that patch as essential. We explore this non-determinism in more detail in Section V-D. A more formal treatment and investigation into causes of non-determinism in rehosted systems is an interest of future work.

V. Experiments

A. Large-Scale Evaluation

The target-centric rehosting approach is intended to help users analyze a given firmware and provide a platform for those users to assess and improve the quality of their rehosting. Though our system is intended for that target-centric approach, we perform experiments to ensure we are at least matching parity with existing large-scale rehosting systems. Furthermore, we wish to evaluate Penguin's performance across a diverse set of firmware vendors.

The first experiment examines how Penguin's automatically-generated configuration performs on a large corpus, particularly to measure the efficacy of extraction and filesystem analysis. This initial analysis and configuration generation is akin to the one-shot analysis of FirmAE, so we utilize FirmAE as a baseline for evaluation and seek to demonstrate the differences in our approach to improve rehosting success across a variety of vendors.

Our large-scale evaluation was performed on a corpus of 13,649 firmware images. The corpus is summarized in Appendix A. Since prior work had used smaller corpa of Linux firmware (1,124 images for FirmAE and 9,379 for firmadyne), we ran FirmAE against our corpus for comparison. Due its unprivileged nature, Penguin was run on an HPC cluster using Singularity [21], [22]. Since FirmAE requires privileges and cannot be used in an HPC environment, the FirmAE results were ran on a single machine over the course of about a month.

A rehosting is considered successful if the emulated system has a process bind to port 80 or 443 (we will discuss this criteria futher in Section V-C). Fig. 3 shows the results of

[4]We observed a mean of 1,344 configuration options per firmware in our experiments. An individual option could be for example, a single NVRAM key.

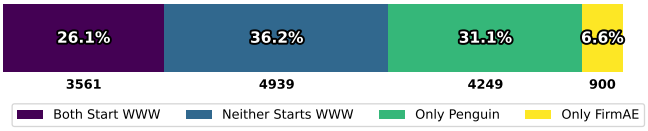

Fig. 3: Large Scale Comparison of Penguin and FirmAE on 13,649 firmware images. WWW server start detected with a bind to port 80 or 443.

our large-scale comparison between FirmAE and Penguin and Fig. 4 shows rehosting success broken down by vendor. From the results, we see that roughly half the firmwares do not start a webserver with Penguin. These failureThese failures can be attributed to various reasons, ranging from failure to extract a filesystem from the provided image to the firmware does not attempt to start a webserver.

Both FirmAE and Penguin have situations where a webserver does not start, some of which can be attributed to non-determinism. We note that Penguin starts 75% more webservers than FirmAE and the fraction of firmwares that start only with Penguin is roughly 4.7x that of firmwares that start only with FirmAE. The table below summarizes the classes of failures for a webserver not starting (percentages are as fraction of failures, not of firmware corpus):

TABLE II: Summary of Failures for Penguin/FirmAE

Class of Failure	FirmAE		Penguin	
Extraction Failure	3342	(36.37%)	2045	(35.02%)
Unsupported Architecture	126	(1.37%)	84	(1.44%)
No Webserver Start	5720	(62.26%)	3710	(63.54%)

Looking at Table II, we see that Penguin's improvements are across all categories of rehosting failure.

B. Non-empty Web Server Response

For this subsection, we consider the 7710 firmwares that bind a webserver with Penguin. Prior work has considered a firmware binding a webserver as a success [2], [3]. While a bind to port 80/443 can be an indication of rehosting health, a service bind alone does not indicate the underlying service is functional. Using the FetchWeb plugin discussed earlier, we check to see if a service returns a non-empty reply by running `wget`. We run the firmware for 600s with support for early termination if we receive a non-empty webserver response or the firmware has completed initialization (using the FICD metric from Pandawan [11]). This simple HTTP interaction is an improved metric over checking for webserver start, but it is far from exhaustive. Further work is needed to explore more exhaustive and useful interactions with a rehosted guest.

Of the 7710 webservers that bind to port 80/443, 5805 (75%) give a non-empty response. That means that 25% of firmwares that bind to port 80/443 likely have a nonfunctional webserver.

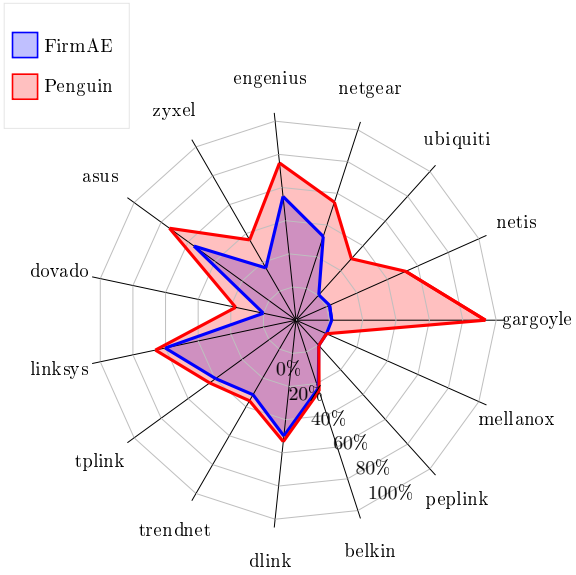

Fig. 4: Rehosting success per vendor for the 15 most prevalent vendors in our dataset. We see that Penguin outperforms FirmAE across most vendors and supports a wider set of vendors, despite not having been designed with expert knowledge for those vendors.

C. Evaluation of Filesystem Analyses

In order to characterize the importance of various analyses to rehosting, we minimized the configurations generated by Penguin using the method described in Section IV-F. We used a non-empty webserver response as our criterion for rehosting success.

The results of configuration minimization are categorized by filesystem analysis in Table III. The minimizer was able to run 5213 of the 5805 firmwares. We observe that 3697 (70.92%) of those firmwares did not require any additional analyses beyond what is essential to run firmware in Penguin. Of the optional analysis features we see that `Library Functions` is the most impactful. The `Library Functions` analysis provides vendor-specific names for functions like `nvram_init`. We observe that providing correct nvram keys is required for only ∼3.7% of the firmwares to give a healthy webserver response, but having correct initialization is required for 16% of firmwares. We note that once one starts to interact with a system more thoroughly, what is important for a rehosting changes (i.e., higher fidelity can be necessary).

Looking at the vendor-specific breakdowns of each analysis type, we see significant variation in the analyses required. While some vendors require little to no additional analyses (Gargoyle, TPlink), ASUS firmwares rely heavily on library functions (due to custom naming of `nvram_` functions as noted in prior work [3]), 91.80% of Netis firmwares require shims that prevent `halt`, and 21.4% of Netgear devices require NVRAM keys. Furthermore, within vendors like Netgear and Belkin, there is notable diversity. This variation across vendors supports the case for deriving intervention information

TABLE III: Filesystem analyses required by the firmwares that give a non-empty webserver response along with vendor-specific distribution. Firmware may require more than one analysis feature. Analyses are described in Table I and Appendix C.

Analysis	Overall	Top Vendors								
		ASUS	Netgear	TPlink	Ubiquiti	Linksys	Dlink	Trendnet	Netis	Belkin
Sample Size	5213	2100	861	431	200	115	111	61	61	23
`ForceWWW`	18.0%	30.0%	3.0%	6.7%	100.0%	2.6%	0.9%	0.0%	0.0%	21.7%
`Library Functions`	15.7%	22.7%	31.4%	0.0%	0.0%	9.6%	12.6%	27.9%	4.9%	65.2%
`NVRAM`	3.7%	0.2%	21.4%	0.0%	0.0%	0.9%	0.0%	0.0%	0.0%	21.7%
`Shims`	1.7%	0.1%	1.7%	0.0%	0.0%	0.0%	0.0%	9.8%	91.8%	0.0%
`Pseudofiles`	0.3%	0.1%	0.5%	0.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
`Network Interfaces`	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	21.7%
`Missing Files/Dirs`	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

from analyses as opposed to hardcoding expert knowledge.

We remind the reader that our simple webserver interaction is not representative of all functionality a security analyst would require from a rehosted system. Once one starts interacting more extensively with a system, features like correct NVRAM keys can become critical. We cover a case study of a system with a non-empty, but poorly performing webserver in Section VI-A.

Since we observe in Table III that many firmwares do not use analyses beyond our default extraction, we were curious to see the impact of minimization on configuration size. In Fig. 5, we calculate the compression ratio of the minimized configuration to the default configuration generated from running all analyses. To keep experiments tractable, configuration optimization occurs at the configuration patch granularity, so quantization effects are present. Still, this minimization demonstrates that users can focus on the salient transformations to get their rehostings to function vs. a one-size-fits-all approach.

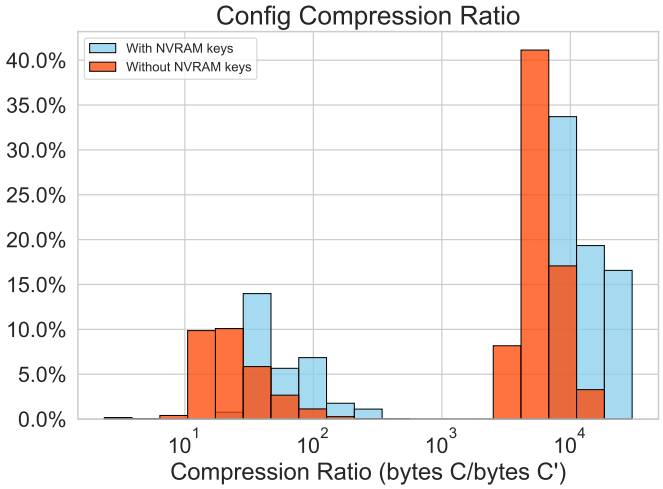

Fig. 5: Compression ratio of default configuration to minimized configuration (for webserver non-empty response). Since NVRAM analysis generates a large number of potential keys, we consider the compression ratio both with and without NVRAM keys.

D. Non-determinism

As noted in Section IV-F, we observed non-determinism in healthy webserver respones and ran each configuration up to ten times. In addition to observing ten successive failures before marking a system as failed we recorded the number of attempts before a given firmware succeeded. We average the number of attempts per firmware and show the distribution in Fig. 6.

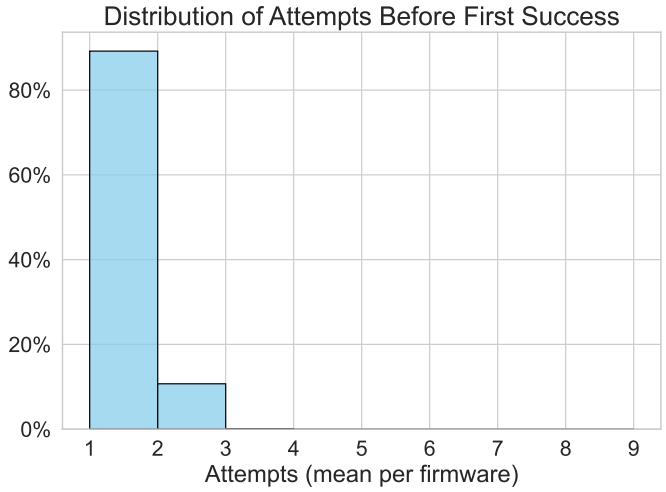

Fig. 6: Histogram showing average number of attempts before a run succeeded with a required filesystem analysis. Each data point is the mean of features per firmware image. I.E., the "2" bin represents the firmwares with an average of 2 attempts per required feature.

From Fig. 6, we see that most runs succeed in one attempt. However, roughly 15% of firmwares require 2+ attempts on average. Fortunately, the tail falls off quickly so future experiments can focus on a smaller number of re-executions.

VI. Case Studies

Below, we discuss case studies of systems that we have rehosted with Penguin. In each example, some target-specific information is needed to successfully rehost the system and Penguin not only makes those transformations simple to apply but also is used to identify which transformations are required.

A. StrideLinx Industrial VPN

The StrideLinx Industrial VPN is a commercial remote access VPN solution. It is an ARM-based Linux 3.4.96 system with support for Distributed Switch Architecture (DSA) hardware, a temperature sensor, real-time clock, and serial communication. The firmware is designed to employ services for `snmpd` and a web management interface among other services.

Penguin is able to identify the architecture and endianness of the system as well as the init program. Its static analysis is able to identify network interfaces and psueodofiles criticial to successful firmware operation. In particular, the `/dev/dsa` device, which manages the Distributed Switch Architecture hardware must exist and return non-erroring `ioctl`s. On its initial run of the system, Penguin is able to run the standard set of services including the web management server (this would satisfy a non-empty response in Section V). However, when opened in a browser it is clear that the web server is not running correctly as it gives errors indicating that the `sxid` (which identifies the specific router variant) is incorrect.[5] This value as well as the `sxserno` (serial number) are arguments expected to be provided to the kernel on the command line from the bootloader. Penguin is able to detect through the dynamic string search described in Section IV-E that the environment variables are referenced and missing. We identify candidate values using the string search results. On the next run of the system, we are able to provide correct values for these environment variables and the web interface is usable. An example of the Penguin configuration options used for this system is in Appendix B.

B. TRENDnet TV-IP201

The TRENDnet TV-IP201 is a wireless network camera server with audio capabilities. It is a MIPS-based Linux 2.4.18 system with RALink WiFi and a USB Camera. The firmware is designed to employ services for web management, upnp, and portmap.

Penguin is able to identify the architecture and endianness of the system as well as the init program. Penguin's suite of static analyses identify several correct network interfaces and 19 pseudofiles including those representing the serial and video interfaces. On the initial run of the system, Penguin is able to run several services including `upnp` and `portmap`, but not the web server. The web server fails because the system is missing the `adm0` network interface. Normally, this interface would be initialized by the kernel module associated with the RALink WiFi device. When looking at Penguin's runtime analysis output, the command `/sbin/iff_get -i adm0` was one of 13 commands executed during startup (and the second-to-last). An analyst can use this information to add the `adm0` interface and rehost the system.

[5]A metric such as a non-empty webserver response would mark this rehosting as healthy, despite its unusable interface. This example highlights the need for more thorough interactions to assess rehosting quality.

C. D-Link DNS320 NAS

The D-Link DNS320 NAS is a consumer-grade Network Storage device. It is an ARM-based Linux 2.6.31 system with hardware support for WiFi and multiple Hard Drives. The firmware is designed to employ services for web management and `smb` among other services. The system is EOL and has widely reported vulnerabilities. In particular, CVE-2024-3273 and CVE-2024-3272 impact the web server and received media attention.

After identifying the architecture, endianness and init program Penguin runs static analyses to identify network interfaces critical to its proper functioning as well as several pseudofiles required for the system to operate. With its initial configuration, Penguin is able to run the standard set of services including the web server. However, as additional services are enabled, the system begins to fail. The system fails because it begins to interrogate emulated hardware for hard drives. While we could have added support for the underlying hard drives through complex pseudofile modeling we chose instead to disable the firmware's ability to disrupt the system by replacing the `shutdown` and `killall` scripts with a shim script that exits without error. With these changes in place we are able to interrogate the web server and other services. This allows us to examine and better understand the target CVEs.

VII. Limitations and Future Work

Currently, our system requires a human-in-the-loop for rehosting. The next step in improving this work would be automation around the configuration generation process. Since our system has dynamic analysis and measurement capabilities, we could propose transformations based on failures and use health metrics to guide the creation of rehosting configurations.

Filesystem extraction plays a large role in the rehosting process. We believe that focusing on a filesystem extractor will be a useful way to improve rehosting success, especially if we can pair that extractor with filesystem analyses. One such opportunity is to extend our *mount* analysis and integrate it with secondary (i.e., non-root) filesystems identified by our extractor. After observing a firmware attempt to mount a secondary filesystem to a given mount point, and potentially failed accesses within the mount point, new configurations could be generated to specify that various secondary filesystems be placed at the mount point.

Finally, we would like to integrate Penguin with Greenhouse [1], a *single-service* rehosting platform. Greenhouse iteratively refines the environment run by a single-service from an embedded system, optionally using a whole-system rehosting from FirmAE as a source of information. As Penguin outperforms FirmAE and allows a system to customize the rehosting environment, we believe integrating Penguin with Greenhouse could improve Greenhouse's ability to rehost single-services from firmware.

VIII. Conclusions

We present Penguin, a target-centric rehosting platform for embedded Linux firmware. Penguin considers a firmware rehosting as a repackaged filesystem with a configuration that specifies transformations to be applied to that filesystem and the runtime environment. This rehosting model of "configuration not code" allows users to iteratively improve, share, and reproduce their rehostings. As a design goal, Penguin runs in unprivileged environments such as HPC clusters. A variety of static and dynamic analyses can be performed to assess rehosting health, study the guest system, and streamline its operation. In particular, we introduced the novel VPNguin utility, which provides and automates robust network connections to services listening inside the rehosted system. In a large-scale evaluation, Penguin outperforms prior work on a diverse corpus of 13,650 Linux firmware images, with improvements across all vendors.

Availability

Penguin code is available at https://github.com/rehosting/penguin

Acknowledgments

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Dept of the Army under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Dept of the Army. Delivered to the U.S. Government with Unlimited Rights, as defined in DFARS Part 252.227-7013 or 7014 (Feb 2014). Notwithstanding any copyright notice, U.S. Government rights in this work are defined by DFARS 252.227-7013 or DFARS 252.227-7014 as detailed above. Use of this work other than as specifically authorized by the U.S. Government may violate any copyrights that exist in this work.

The authors acknowledge the MIT Lincoln Laboratory Supercomputing Center for providing HPC and database resources that have contributed to the research results reported within this paper/report.

References

[1] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, 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.

[2] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *NDSS*, 2016.

[3] 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 *ACSAC*. ACM, 2020.

[4] 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.

[5] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, "Sok: Enabling security analyses of embedded systems via rehosting," in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 687–701.

[6] 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 rehosting," in *RAID*, 2019.

[7] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *USENIX Security*, 2020.

[8] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM CSUR*, 2021.

[9] C. Heffner, "Binwalk: Firmware analysis tool," Open Source Software, 2013, a tool for searching a given binary image for embedded files and executable code. Designed for firmware analysis. [Online]. Available: https://github.com/ReFirmLabs/binwalk

[10] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu *et al.*, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 442–454.

[11] I. Angelakopoulos, G. Stringhini, and M. Egele, "Pandawan: Quantifying progress in linux-based firmware rehosting," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5859–5876.

[12] Q. Kaiser, "Blackalps 2022: Firmwares are weird. a year long journey to efficient extraction," https://www.youtube.com/watch?v=PZ_gw85PcgY, 2022.

[13] E. Bendersky, "pyelftools," https://github.com/eliben/pyelftools, 2012, a Python library for working with ELF files and DWARF debugging information.

[14] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *ACSAC PPREW*, 2015.

[15] L. Craig, A. Fasano, T. Ballo, T. Leek, B. Dolan-Gavitt, and W. Robertson, "Pypanda: Taming the pandamonium of whole system dynamic analysis," in *Workshop on Binary Analysis Research (BAR)*, vol. 2021, 2021, p. 21.

[16] G. F. Lyon, "Nmap: Network Mapper," Open Source Software, 2023, a security scanner used to discover hosts and services on a computer network, thereby building a "map" of the network. [Online]. Available: https://nmap.org/

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[18] S. Hajnoczi, "virtio-vsock: Zero-configuration host/guest communication," in *KVM Forum*, 2015.

[19] J. Wiesböck, J. Naab, and H. Stubbe, "Virtio-vsock-configuration-agnostic guest/host communication," *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM)*, pp. 73–78, 2019.

[20] BusyBox Contributors, "BusyBox: The Swiss Army Knife of Embedded Linux," Open Source Software, 2023, busyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. [Online]. Available: https://www.busybox.net/

[21] A. Reuther, J. Kepner, C. Byun, S. Samsi, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, L. Milechin, J. Mullen, A. Prout, A. Rosa, C. Yee, and P. Michaleas, "Interactive supercomputing on 40,000 cores for machine learning and data analysis," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.

[22] Sylabs Inc., "Singularity: Container for High-Performance Computing," Software, 2023, singularity is a container platform designed to be simple, fast, and secure. It is specifically tailored for high-performance computing (HPC) environments. [Online]. Available: https://sylabs.io/singularity/

APPENDIX

A. Corpus Summary

The firmware corpus used in our large-scale evaluation contained 13,649 images obtained from vendor websites and represents firmware from 69 different vendors. Architecture identification was performed by Penguin.

Architecture	Count	Percentage
ARM	4347	31.8%
MIPS Big Endian	3489	25.6%
MIPS Little Endian	2675	19.6%
AARCH64	764	5.6%
MIPS64 Big Endian	161	1.2%
PowerPC	84	0.6%
x86	62	0.5%
x86_64	10	0.1%
Uknown	2057	15.1%

Note: "Unknown" corresponds to firmware where extraction or architecture identification failed.

Vendor	Count	Percentage
ASUS	3927	28.8%
Netgear	2893	21.2%
Gargoyle	1504	11.0%
TPlink	1144	8.4%
Ubiquiti	839	6.1%
DLink	334	2.4%
Trendnet	333	2.4%
Peplink	279	2.0%
Zyxel	222	1.6%
Linksys	219	1.6%
EnGenius	203	1.5%
Belkin	182	1.3%
Dovado	164	1.2%
Netis	154	1.1%
Other	1252	9.2%

Note: "Other" includes all vendors that represented less than 1% of our corpus.

B. Configuration Example

Below is an example configuration patch for the StrideLinx Industrial VPN Router as discussed in Section VI-A. The configuration includes setting an environment variable telling the firmware what router variant is to be used, which can be found using the Dynamic String Search in Section **??**. The configuration also contains the result from our automated analysis that finds a missing pseudofile, `/dev/dsa` and models that device to return as if all `ioctl`s on it succeed.

```
env:
  sxid: 0150_5MS-MDM-1
# Below is generated automatically
pseudofiles:
  /dev/dsa:
    ioctl:
      "*":
        model: return_const
        val: 0
```

C. Filesystem Analysis Details

Architecture When identifying the target architecture, Penguin recursively scans the provided filesystem. This analysis prefers false negatives in order to heavily reduce likelihood of false positives. Both the file path and the contents of the file must look like that of an ELF binary. For the path check, a binary must either be in a standard directory for executables (`/sbin`, `/bin`, `/usr/sbin`, or `/usr/bin`) or be a shared object or kernel module (have a file extention of `.so`, `.ko`, or `.so.*`). The file must then be successfully be parsed with Python's elftools library [13]. If all these checks pass we tally the architecture noted in the header. We then select the most common architecture, taking a preference for 64-bit architectures to account for 64-bit firmwares that utilize backwards compatibility for 32-bit binaries as the 32-bit binaries may outnumber their 64-bit counterparts.

This style of firmware architecture detection extends the work of Firmadyne [2], which similarly uses the headers of ELF binaries present in the firmware to determine the architecture. Unlike Firmadyne (and the subsequent FirmAE) we take into consideration that filesystems may, for a variety of reasons, include binaries intended for other devices. Since Firmadyne is only concerned with the first binary it can identify, this can result in incorrect results when attempting to ascertain the architecture. Similarly, small inaccuracies can occur due to usage of the `file` utility as a means of architecture detection, as any non-ELF file which includes the name of an architecture in either its path or its metadata can result in an incorrect architecture being selected.

Init Our default behavior when the init cannot be determined with certainty is to apply a heuristic in which we search for a variety of path-based conventions for the naming of init scripts. This includes the existence of strings such as `init` or `start` in the filename, the exclusion of common false positives (`inittab`, directories, restart scripts), and finally ensuring the candidate is a valid init (marked as executable, resolvable if a symlink). After filtering down the candidates we sort by naming conventions indiciating execution order (`preinit` over `init` over `rcS`) and use path length as a heuristic for a file being sufficiently important and a global part of the system, ensuring scripts in overly nested directories are not choosen over files closer to the root of the filesystem. This approach of fuzzy matching based on convention allows for finding a much larger set of `init` candidates over Firmadyne,

File Path	Startup Command
/etc/init.d/uhttpd	/etc/init.d/uhttpd start
/usr/bin/httpd	/usr/bin/httpd
/usr/sbin/httpd	/usr/sbin/httpd
/bin/goahead	/bin/goahead
/bin/alphapd	/bin/alphapd
/bin/boa	/bin/boa
/usr/sbin/lighttpd	/usr/sbin/lighttpd -f /etc/lighttpd/lighttpd.conf

TABLE IV: Web Server Startup Commands Added by ForceWWW.

which uses a much simpler approach of customizing the kernel to run each item in a short, hardcoded list of candidates in sequence. Our approach also eliminates possible reentrancy bugs caused by running multiple `inits`, some of which may in turn execute each other, such as if `/sbin/init` runs `/bin/init`. In comparison, FirmAE attempts to rectify this by using strings from the kernel in order to find hardcoded boot arguments to determine the `init` path.

ForceWWW Penguin includes an intevention `ForceWWW` which includes some light static analysis meant to replicate FirmAE's intervention for unexecuted web servers. Our implementation is intended to be an exact replication to ensure this is not a factor in our evaluation. This approach involves searching for members of a fixed set of possible web servers and, if present, the corresponding command will be included in our startup process. A list of these web servers and commands is included in Table IV.

Missing Directories Oftentimes rehosting is performed with partial copies of the device firmware, such as with update blobs that include only the files and directories that have changed since the original version of the firmware. In order to handle those cases Firmadyne introduced a hardcoded list of directories. FirmAE extends that approach by both adding to this list of directories and additionally searching for paths within executables to ensure the needed directories are present. Penguin reimplements all those interventions with minimal changes to the logic behind them, with the most notable change being that modifications to the filesystems are represented as entries in the configuration file rather than actually modifying the filesystem directly, which is beneficial for later iterating on a rehosting. In addition an intervention is introduced to discover mountpoints used in shell scripts in order to ensure mount targets are present.

Missing Files Similarly many firmwares depend on a few common files we attempt to provide if they are not already present, such as ensuring a bash shell is present and timezone information is present. We apply an intervention which ensures files shown to be commonly needed across different firmwares are included. The `/etc/hosts` file is also ammended if needed to ensure it has an entry for `localhost`. These are all reimplementations of Firmadyne's equivelant interventions. Included below is an example of the resulting changes applied for fixing a missing directory (`/tmp/lock`) and missing timezone configuration (`/etc/TZ`).

```
static_files:
  # ...
  /tmp/lock:
    type: dir
    mode: 493
  /etc/TZ:
    type: inline_file
    contents: EST5EDT
    mode: 493
  # ...
```

Shims Penguin utilizes shims as a means of intercepting entire utilities by replacing them with symlinks to desired utilities in the filesystem. The original copies are stored so that either an analyst or the provided replacement can run them if needed. In some situations shims are used to prevent unwanted behavior such as restarting or loading incompatible kernel modules while continuing gracefully. Shims are also used for performance improvements (cryptography), logging (bash), and behavior modification (`getenv`, `setenv`).

Pseudofiles While our static analysis plugins do not include any analysis that attempts to properly model pseudofile behavior, we do replace any statically-present pseudofiles, such as device files, with emulated pseudofiles with default behavior that has reads return zeroed buffers while writes succeed but the contents are discarded. In comparison Firmadyne offers many default devices via kernel modifications, but does not provide tailored pseudofiles as a part of its execution environment.

```
pseudofiles:
  /dev/acos_nat_cli:
    read:
      model: zero
    write:
      model: discard
    ioctl:
      '*':
        model: return_const
        val: 0
```

ABI In order to perform interception of library calls, libinject must know which ABI the dynamic library it is intercepting uses. In order to handle this Penguin scans the filesystem for copies of libc in order to observe all ABIs used

by dynamically linked binaries. From there libinject checks a combination of `e_flags` (MIPS) and `.ARM.attributes` (ARM) in order to determine the ABI used by libc. A copy of libinject is built for every ABI observed this way and the corresponding copy is provided via `LD_PRELOAD`. This mechanism is also utilized for enabling library call interception on mixed-bitwidth systems, such as on 64-bit x86 firmwares with 32-bit binaries.

Network Interfaces Due to the differences in how Penguin handles networking compared to its predecessors (see Section IV-D2) our approach to discovery of network interfaces is different. We only need to provide interfaces with the expected name for the sake of preventing rehosting failures rather than for actual use. To do this we search for a variety of usage patterns to discover the names of needed interfaces. First we search for paths to files under `/sys/class/net` indicating the expectation of an interface. Then we search strings present in binaries on the filesystem for usage of networking tools including `ifconfig`, `ethtool`, `route`, `netstat`, and more to find examples of information about interfaces being queried.