Towards Automated Dynamic Analysis for Linux-based Embedded Firmware

Daming D. Chen*, Manuel Egele†, Maverick Woo*, and David Brumley*
* Carnegie Mellon University
{ddchen, pooh, dbrumley}@cmu.edu
† Boston University
{megele}@bu.edu

***Abstract*—Commercial-off-the-shelf (COTS) network-enabled embedded devices are usually controlled by vendor firmware to perform integral functions in our daily lives. For example, wireless home routers are often the first and only line of defense that separates a home user's personal computing and information devices from the Internet. Such a vital and privileged position in the user's network requires that these devices operate securely. Unfortunately, recent research and anecdotal evidence suggest that such security assumptions are not at all upheld by the devices deployed around the world.**

A first step to assess the security of such embedded device firmware is the accurate identification of vulnerabilities. However, the market offers a large variety of these embedded devices, which severely impacts the scalability of existing approaches in this area. In this paper, we present FIRMADYNE, the first *automated* dynamic analysis system that specifically targets Linux-based firmware on network-connected COTS devices in a scalable manner. We identify a series of challenges inherent to the dynamic analysis of COTS firmware, and discuss how our design decisions address them. At its core, FIRMADYNE relies on software-based full system emulation with an instrumented kernel to achieve the scalability necessary to analyze thousands of firmware binaries automatically.

We evaluate FIRMADYNE on a real-world dataset of 23,035 firmware images across 42 device vendors gathered by our system. Using a sample of 74 exploits on the 9,486 firmware images that our system can successfully extract, we discover that 887 firmware images spanning at least 89 distinct products are vulnerable to one or more of the sampled exploit(s). This includes 14 previously-unknown vulnerabilities that were discovered with the aid of our framework, which affect 69 firmware images spanning at least 12 distinct products. Furthermore, our results show that 11 of our tested attacks affect firmware images from more than one vendor, suggesting that code-sharing and common upstream manufacturers (OEMs) are quite prevalent.

I. Introduction

With the proliferation of the so-called "Internet of Things", an increasing number of embedded devices are being connected to the Internet at an alarming rate. Commodity networking equipment such as routers and network-attached storage boxes are joined by IP cameras, thermostats, or even remotely-controllable power outlets. These devices frequently share certain technical characteristics, such as embedded system on a chip (SOC) designs based on ARM or MIPS CPUs, network connectivity via Ethernet or WiFi, and a wide variety of communication interfaces such as GPIO, I2C, or SPI. Nevertheless, many of these devices are controlled by vendor and chipset-specific firmware that is rarely, if ever, updated to address security vulnerabilities affecting these devices.

Unfortunately, the poor security practices of these device vendors are only further exacerbated by the privileged network position that many of these devices occupy. For example, a wireless router is frequently the *first and only* line of defense between a user's computing equipment (e.g., laptops, mobile phones, and tablets) and the Internet. An attacker that succeeds in compromising such a networking device is able to gain access to the user's network, and can further reconfigure the device to tamper with arbitrary network traffic. Since most vendors have not taken any initiative to improve the security of their devices, millions of home and small business networks are left vulnerable to both known and unknown threats. As a first step towards improving the security of commodity computer equipment, we propose to address the challenge of *accurately identifying vulnerabilities in embedded firmware* head-on.

Previous research on the security of embedded firmware can be categorized based on various analysis approaches. For example, Zaddach et al. [19] perform dynamic analysis by partially offloading execution of firmware to actual hardware. While such an approach is precise, it incurs significant hurdles for large-scale analysis. First, the requirement that the analyst must obtain the physical hardware for the device under test poses a significant financial burden. Second, and more importantly, the manual effort needed to identify and interface with the debugging port on the device places strict limits on the scalability of this technique, especially for consumer equipment that may not support hardware debugging functionality.

In contrast, Costin et al. [8] utilize static analysis techniques to unpack the firmware of embedded devices and identify potentially vulnerable code or binaries inside. While this approach scales to thousands of firmware images, it suffers from the classic trade-offs of static analysis. Namely, either the analysis is very generic and produces a large number of false positives [5], or the analysis is too specific and results in many

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23415

false negatives. Additionally, static analysis techniques based on program analysis usually target a specific problem domain, such as the C, PHP, or Java programming language, or alternatively binary code. Unfortunately, commodity networking equipment typically contains an amalgamation of various programs and scripts, written in a variety of compiled or interpreted programming languages. Oftentimes, custom modifications are even made to the language runtime to cater to the unique requirements of embedded systems.

To overcome the shortcomings of previous work in this area, we leverage software-based full system emulation to enable large-scale and automated dynamic analysis for commodity embedded firmware. Since our approach does not rely on physical hardware to perform the analysis, it scales with additional computational resources. Additionally, our full system emulation approach transparently provides dynamic analysis capabilities, regardless of the programming language used to develop a specific application or script. Furthermore, we inherit the precision of other dynamic analysis techniques—if the analysis finds that a firmware image contains a vulnerability, then it provides actionable results in the form of a successful exploit. Finally, we address a number of challenges that are characteristic for embedded devices, such as the presence of various hardware-specific peripherals, storage of persistent configuration in non-volatile memory (NVRAM), and dynamically-generated configuration files.

We implemented FIRMADYNE to demonstrate our approach to automated dynamic analysis. Using firmware image files distributed on vendor support websites, we automatically unpack the contents to identify the kernel and extract the filesystem. Since the majority of these extracted firmware are Linux-based, we initially focus on support for Linux-based firmware by pre-compiling modified Linux kernels. Using the `QEMU` [4] full system emulator, we are able to boot our instrumented kernels with the extracted filesystem from the original firmware images. In order to collect a dataset of these firmware images, FIRMADYNE includes a web crawler that automatically downloads metadata and firmware images from various vendor websites, which are then fed into the dynamic analysis system.

However, even with full system emulation, an emulated environment must be configured correctly to interact with the network interfaces of the guest firmware. Therefore, our system initially emulates the guest in an isolated network environment, and monitors all network interactions to infer the correct configuration for subsequent analyses. Once this information is collected, FIRMADYNE will re-configure the emulated environment with the inferred network configuration, enabling network interaction between the emulated guest firmware and the analysis host.

With the aid of our analysis and introspection capabilities, we identified 14 previously-unknown vulnerabilities for which we were able to manually develop proof-of-concept exploits. Of these, across our dataset of 23,035 firmware images gathered from 42 device vendors, we identified 69 vulnerable firmware images spanning at least 12 distinct products from the 9,486 firmware images that were successfully extracted. Since the process of emulating and testing firmware images in

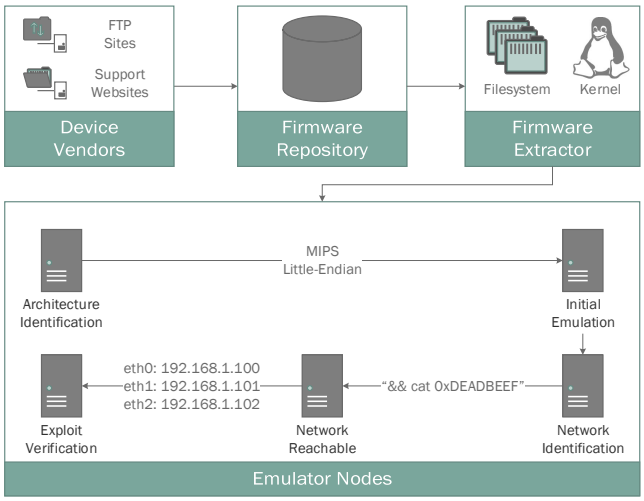

Fig. 1: Architectural diagram of FIRMADYNE showing the emulation life-cycle for an example firmware image, as described in §II-A.

FIRMADYNE is automated, it was straightforward to integrate a subset of the existing exploits from the popular Metasploit Framework [2].

Using these results, we observe that the most prolific exploit affects the firmware of up to five different vendors, and the most effective exploit affects 10% of all network inferred firmware images in our dataset. While code-reuse of vulnerable open-source applications is one explanation, our attacks also affect applications whose source is not publicly available, suggesting that code-sharing and common upstream manufacturers (OEMs) are quite prevalent.

To summarize, the contributions of this work are as follows:

- We present FIRMADYNE, our implementation of an automated and scalable dynamic analysis technique specifically designed to accurately identify vulnerabilities in Linux-based embedded firmware (§II).
- Our implementation of FIRMADYNE addresses characteristic challenges of embedded systems, such as the presence of hardware-specific peripherals, usage of non-volatile memory (NVRAM), and creation of dynamically-generated files (§IV).
- We gathered a dataset of 23,035 firmware images downloaded from 42 different vendors, and evaluated FIRMADYNE on the 9,486 firmware images that were successfully extracted, using a set of 14 previously-unknown and 60 known exploits (§V).
- In support of open science, we make our system available to the research community under an open-source license to encourage further research into embedded systems. For more information, please see https://github.com/firmadyne/.

II. Overview

In this section we describe the design of various components that comprise FIRMADYNE, and our motivations for such an architectural design.

A. Components

As depicted in Fig. 1, FIRMADYNE consists of four major components.

1) Crawling Firmware: The first and largely independent component is a web crawler, which downloads firmware images from vendor websites. At present, we support 42 device vendors (see §IV-A). We manually wrote parsing templates for each of these websites, allowing us to distinguish between firmware images and other binary content. This targeted crawling effort provided us with metadata for each gathered firmware image, including information such as the build date, release version, and links to Management Information Base (MIB) files for the Simple Network Management Protocol (SNMP). Such metadata proved useful for our automated analyses and exploit development (see §V-B3). For dynamic websites that were difficult to crawl automatically, we instead crawled the vendor's FTP site, at the expense of no metadata.

2) Extract Firmware Filesystem: In the second step, FIRMADYNE uses a custom-written extraction utility built around the `binwalk` [1] API to extract the kernel (optional) and the root filesystem contained within a firmware image (see §IV-B).

3) Initial Emulation: Once a filesystem is extracted, FIRMADYNE identifies the hardware architecture of the firmware image; in Fig. 1, we have chosen `MIPS Little-Endian` as an example. Then, our system uses a pre-built Linux kernel in an instance of the `QEMU` full system emulator that matches the architecture, endianness, and word-width of the target firmware image. Currently three combinations are supported: little-endian ARM, little-endian MIPS, and big-endian MIPS. An initial emulation is performed to infer the system and network configuration, shown as three IP address assignments to `eth0`, `eth1`, and `eth2` for the example in Fig. 1. This is achieved by intercepting system calls to the filesystem, networking, and other relevant kernel subsystems.

4) Dynamic Analysis: The forth and final step can be repeated for any dynamic analysis supported by FIRMADYNE. To this end, the environment is dynamically reconfigured to match the expectations of the firmware image (see §IV-C) as inferred in the previous step. Note that FIRMADYNE is designed for easy extensibility to include new dynamic analyses or exploits. The results of each individual analysis are aggregated in a database for ease of inspection. In the example above, shown in Fig. 1, a command injection vulnerability is being tested on the target firmware image.

To illustrate this versatility, we have developed three vulnerability detection passes, which are able to assist in finding vulnerabilities and precisely identify whether a given exploit succeeds by monitoring events from our instrumented kernel. These passes helped us detect 14 previously unknown vulnerabilities, which were automatically confirmed to affect 69 firmware images, based on proof-of-concept exploits that we developed (see §V-B). We further demonstrate the flexibility of FIRMADYNE by seamlessly integrating 60 known exploits mostly from the popular Metasploit [2] exploit framework. In total, both vulnerability types affect 887 firmware images from our dataset.

B. Motivation

Dynamic analysis targeting embedded system firmware addresses a variety of design points in the abstraction hierarchy of embedded systems. We discuss a selection of potential vantage points for such analysis, illustrate challenges and shortcomings, and argue why dynamic analysis based on full system emulation is the most promising approach to tackle this challenge.

1) Application-Level: Perhaps the most straightforward approach is to statically extract application-specific data, and execute it natively with a supported application. For example, it is possible to copy the webpages served by a web server within an embedded system, and serve the content using a regular web server such as Apache. Unfortunately, this approach has multiple drawbacks that are incompatible with our design goal of creating a generic platform for dynamic analysis of embedded firmware.

An analysis of the firmware images in our dataset shows that many of these contain webpages which rely on non-standard extensions to server-side scripting languages (e.g., `PHP`) for access to hardware-specific functionality, such as NVRAM values. For example, hundreds of images in our dataset make use of the custom functions `get_conf()` in `PHP` and `nvram_get()` in `ASP.NET` to obtain device configuration values. Unfortunately, this functionality is a custom addition to the web server that is not supported by their upstream open-source counterparts. Additionally, other firmware images do not place these webpages on the filesystem, but instead embed their HTML content within the binary of a custom web server.

Finally, an analysis approach focused on application-data can only detect vulnerabilities within the application-specific data (e.g., command injection vulnerabilities in `PHP` files), but not those present within the original application or other system components.

2) Process-Level: Another feasible approach for analyzing embedded systems is to emulate the behavior of individual processes within the context of the original filesystem. This can be achieved by executing `QEMU` in user-mode as a single process emulator, constrained using `chroot` to the original filesystem. Thus, one could simply launch the original web server from the firmware image in `QEMU`, and then that process would emulate the router web interface.

Unfortunately, this approach only partially obviates the concerns mentioned above. While an application would be able to execute within the context of the filesystem, specific hardware peripherals (e.g., NVRAM) are still unavailable. As a result, when an application attempts to access the NVRAM peripheral via `/dev/nvram`, it will likely terminate in error.

Similarly, minor differences in the execution environment can have a significant effect on program behavior. For example, the `alphafs` web server used by multiple firmware images

verifies hardware-dependent product and vendor IDs before accessing NVRAM. If these values are not present at predetermined physical memory addresses, the web server ceases operation and terminates with an error message. To this end, the web server uses the `mmap()` system call to access memory via `/dev/mem`, and checks specific offsets for the `ProductID` and `VendorID` of supported EEPROM chips.

Emulating such behavior with a user-mode emulator would be complex, as the emulator would need to track file handles and system calls that map memory to determine program behavior. Then, the emulator would need to identify the semantic definition of various memory addresses, and replace the values as appropriate (e.g., a valid `ProductID` and `VendorID`).

Additionally, due to limited write cycles on the primary storage device, many firmware images mount a temporary memory-backed filesystem at boot for volatile data. This filesystem is mounted and generated dynamically. As a result, the directories `/dev/` and `/etc/` may be symbolic links to subdirectories within the temporary filesystem, thus appearing broken when examined statically. For example, the firmware for the D-Link DIR-865L wireless router uses a startup script to populate configuration for applications, including the `lighttpd` web server. This configuration file is then passed to the web server binary with the `'-c'` command line argument. As a result, simple dynamic emulation of the `lighttpd` binary will fail, even with the original filesystem in place.

These types of environmental differences can have a significant effect on the presence of vulnerabilities. For example, many information disclosure vulnerabilities can simply be fixed with proper access control policies. Likewise, the effect of a directory traversal attack on a web server can be greatly affected by the system configuration.

Although this approach is clearly more accurate than the previous approach, it should be apparent that it suffers from a number of shortcomings due to low emulation fidelity. Without precise knowledge of the runtime system environment, the host environment can inadvertently affect dynamic analysis of individual processes by altering program execution.

3) System-Level: In comparison, a system-level emulation approach is able to overcome the aforementioned challenges. Expected interfaces to hardware peripherals will be present, allowing their functionality to be gracefully emulated. Accurate emulation of the system environment permits dynamically-generated data to be created in the same manner as on the real device. All processes launched by the system can be analyzed, including various daemons responsible for protocols such as HTTP, FTP, and Telnet.

During the design process, we explicitly chose full system emulation as the basis for FIRMADYNE for these reasons. By leveraging the built-in hardware abstraction provided by the kernel, we replace the existing kernel with our modified kernel specifically designed and instrumented for our emulation environment. Then, in conjunction with a custom user-space NVRAM implementation, we boot the extracted filesystem and our pre-built kernel within the `QEMU` full system emulator. Otherwise, booting the original kernel would result in a fatal execution crash, since it is only compiled to support a specific hardware platform. Using the system boot sequence provided by the `init` and `rcS` binaries on the original filesystem, we are able to initialize user space to a state consistent with the original device, despite platform changes.

Our results (see §V-A) show that this approach is successful for initial emulation of over 96.6% of all Linux-based firmware images in our dataset. This is likely due to the stable and consistent interface between user-space and kernel on Linux systems, with the exception of custom `IOCTL`'s introduced by vendor-specific kernel modules. In fact, Linux kernel developers will revert kernel changes that break backwards-compatibility for user-space applications; for example, programs built for pre-0.9 (pre-1992) kernels will still function correctly even on the latest kernel releases.[1]

However, this does not hold for kernel modules; indeed, one of the drawbacks of our current implementation is the lack of emulation support for out-of-tree kernel modules located on the filesystem and so differences in kernel version may result in system instability. Nevertheless, our dataset shows that such support is generally not necessary, as more than 99% of all out-of-tree kernel modules within the firmware images in our dataset are not useful for our system (§V-A3). One major reason is because newer kernels, such as those that we build, provide in-tree equivalents for functionality previously developed as out-of-tree extensions. In particular, 58.8% of out-of-tree kernel modules are used to implement various networking protocols and filtering mechanisms that may not have been present in older kernels, and 12.7% provide support for specific hardware peripherals. For example, older 2.4-series mainline kernels lacked `netfilter` connection tracking and NAT support for various application-specific protocols such as TFTP, G.323, and SIP, which became available in-tree around kernel version 2.6.20. In comparison, the third-party NetUSB kernel module, which was recently identified to contain a remotely-exploitable buffer overflow vulnerability, comprises less than 0.2% of all kernel modules from our dataset (§V-A3).

III. Concept

This section provides an overview of the concept behind our dynamic analysis framework for Linux-based firmware images. For specific challenges encountered and implementation details, please see §IV.

A. Architecture

As shown in Fig. 1, our system features a firmware repository server that is used to store the binaries corresponding to each firmware image and a database that keeps track of information pertaining to each firmware image. This includes the extraction status, architecture, brand of each image, as well as each file within a given image.

A set of virtualized worker nodes are used to extract the root filesystem and kernel (optional) from each firmware image. Throughout this process, the database is updated with the current experiment progress. If the extraction is successful, the firmware repository will cache the archived filesystem. Next, these workers enter the learning phase, where firmware

[1] https://www.kernel.org/doc/Documentation/stable_api_nonsense.txt

images are assigned a default configuration and the networking interactions are recorded. This allows our system to infer the correct emulated network environment. Finally, the workers enter the analysis phase, where each firmware image is emulated with the inferred network environment, and individual analyses are performed.

B. Acquisition

In order to gather a representative dataset of firmware images, we developed a custom web crawler. Instead of using a blind crawling methodology, we wrote smart parsers for the support pages of each of our 42 preselected vendors (§A). This allowed us to distinguish between firmware updates and undesired binaries such as drivers, configuration utilities, and other binaries. Additionally, with a better semantic understanding of the target website, we recovered important metadata about each firmware image, such as vendor, product name, release date, version number, changelog, etc.

Where applicable, this was supplemented with probable firmware images that were mirrored from the FTP websites of target vendors. Although this latter source of firmware was less rich in metadata, it provided us with additional binaries that were not directly accessible for all end-users, including betas and test binaries with limited releases. A few brands of firmware images, for which it was difficult to automate, or when the vendors did not provide direct firmware downloads for end-users, were gathered by hand.

C. Extraction

We developed a custom extraction utility using the API of the `binwalk` firmware extraction tool to recover the root filesystem and (optionally) kernel from each firmware image. These were normalized by storing them as compressed `TAR` archives within our firmware repository.

D. Emulation

Once the root file system has been extracted from a firmware image, FIRMADYNE performs a series of analysis steps to infer the system configuration expected by the firmware image.

First, we examine the `ELF` header of binaries located within the extracted root filesystem to identify the target architecture and endianness. For each firmware image, we use the `QEMU` full system emulator for the corresponding architecture to boot the extracted filesystem with a matching kernel. Currently, we have pre-compiled kernels for ARM little-endian, MIPS little-endian, and MIPS big-endian platforms, as our data shows that these architectures constitute 90.8% of our dataset (§V-A1).

Next, during the initial emulation phase, the system is executed in a special "learning" mode, in which our modified kernel records all system interactions with the networking subsystem, including IP address assignments for individual network interfaces.

Finally, after collecting this information, FIRMADYNE enters the actual emulation phase, in which a matching network environment is configured to communicate with the emulated firmware. To verify successful network configuration, FIRMADYNE launches the emulated firmware image and performs a series of network connectivity checks.

E. Automated Analyses

We implemented three basic automated analysis passes within our dynamic analysis framework in order to demonstrate the effectiveness of our system. These contributed to our detection of 14 previously-unknown vulnerabilities that affect 69 firmware images, and a total of 74 vulnerabilities that affect 887 firmware images (see §V).

IV. Implementation

This section discusses the implementation behind each of the components mentioned in §II-A and §III.

A. Acquisition

Our custom web crawler was developed using the `Scrapy` framework, with an individual spider written for each of the 42 vendors in our dataset. To increase representativeness, our dataset includes vendors for networking products ranging from consumer to professional network equipment, such as IP cameras, routers, access points, NAS's, smart TV's, cable modems, satellite modems, and even third-party or open-source firmware. We created individual parsers for the support pages of each vendor using XPath selectors to enumerate and expand specific elements of input webpages. In addition, we also attempted to crawl multiple geographic locations of each vendor's website, including United States (English), China (Chinese), Russia (Russian), European (English), Germany (German), and South Africa (English).

Some vendors that made heavy use of dynamically-generated content on their websites, such as D-Link and ZyXEL, were crawled through their FTP mirror site instead. Only FTP files that appeared relevant were downloaded, which was generally limited to the following filename extensions: `img`, `chk`, `bin`, `stk`, `zip`, `tar`, `sys`, `rar`, `pkg`, and `rmt`. Other vendors, such as Cisco, which made their website difficult to automatically crawl, or limited most firmware downloads to customers with valid support contracts, were manually crawled. Supported metadata fields that were automatically gathered from vendor websites include the product name, vendor name, version, build, date, changelog, SNMP MIB file, source code URL, and firmware image URL. This allows us to distinguish between multiple products that share the same firmware image, since we deduplicate downloaded firmware image binaries. However, not all vendors had such information available, and no metadata was available for vendors crawled through FTP or manually.

B. Extraction

Through manual experimentation, we determined that the built-in recursive extraction mechanism ("Matryoshka") within `binwalk` was insufficient for our purposes. In particular, this extraction was vulnerable to path explosion by attempting to recursively extract compressed data from within an ELF executable or every file within a filesystem, and not guaranteed

to terminate, especially in the presence of false positive signature matches.

Instead, we developed a custom goal-driven extraction utility using the `binwalk` API that minimized disk space and runtime by terminating when our extraction goals were achieved; namely obtaining root filesystem and (optionally) kernel from within each firmware image. In addition, we implemented a set of heuristics for early detection of non-firmware files, which would otherwise waste computational resources. This included blacklisting input files that were any type of structured binary, including PE32 executables for Windows, ELF executables for Linux, and universal binaries for Macintosh, as well as bytecode and relocatable objects. Other common formats that were excluded included PDF files and Microsoft Office documents, which would otherwise appear as compressed archives that require recursive extraction.

After blacklist verification, the extraction process used a set of priority-ranked signatures that were executed sequentially in the order of confidence. These signatures can be categorized as follows: archive formats, firmware headers, kernel magic or version strings, UNIX-like root filesystems, and finally compressed data. Matches for archive formats or compressed data were then extracted recursively. We verify that UNIX-like root filesystems are successfully extracted by checking for the presence of at least four standard root directories from a subset of the Filesystem Hierarchy Standard[2].

Our method allowed us to reduce the effect of false positive signature matches by prioritizing higher-confidence signature matches (e.g., firmware headers) over more generic signature matches (e.g., compressed GZIP data). For example, if upstream `binwalk` detects compressed data within the kernel image of a firmware image and recursive extraction is enabled, it will waste resources attempting to fully extract this data.

Another improvement that we made to the extraction process was utilizing the third party `jefferson` and `sasquatch` extraction tools for `JFFS2` and `SquashFS` filesystems, respectively, which can be difficult to extract. This is because the userspace extraction utilities provided by filesystem developers, `jffsdump` and `unsquashfs`, frequently fail to extract real-world filesystems of these types.

In part, this is because these user-mode extraction utilities are rarely updated and can lag behind the in-kernel filesystem code in terms of filesystem support. More importantly, many device manufacturers have modified existing compression algorithms or even implemented new compression algorithms for these filesystems, making their variants incompatible with other implementations.

To resolve this problem, other firmware extraction utilities such as `bat` and `firmware-mod-kit` rely on a set of heuristics and precompiled `unsquashfs` binaries gathered from the GPL source code releases for various routers. However, this approach is incomplete and ineffective, as maintainers for these extraction utilities need to manually compile new binaries and implement the appropriate heuristics.

[2]http://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf

In contrast, we utilize tools that are specifically written to extract the contents of these modified filesystems from userspace. `sasquatch`, which was developed by the author of `binwalk`, is designed to support as many modified `SquashFS` implementations as possible by adapting to changes in compression algorithms, and recognizing the structure of `SquashFS` filesystems instead of specific magic strings.

During this process, we identified a number of bugs and made improvements to both `binwalk` and `jefferson`, which were submitted to the respective upstream projects. The majority of our submitted patches have already been merged into the official release, and some are still pending maintainer review.

Although these improvements contribute to our success rates, not all firmware images can be extracted by our current implementation. For example, some vendors only distribute partial firmware images for their products, preventing us from reconstructing the root filesystem. Other vendors distribute firmware images with multiple embedded or partial filesystems, which require additional logic to reassemble partial filesystems, or filesystems mounted on top of one another. Furthermore, other vendors distribute encrypted firmware images, firmware images within a binary updater executable, non-Linux-based firmware images, or Linux-based firmware images with unrecognized filesystems, all of which we do not support. As a result, these images are categorized as unknown in Table II.

C. Emulation

1) NVRAM: From a cursory inspection, at least 52.6% of all extracted firmware images (4,992 out of 9,486) access a hardware non-volatile memory (NVRAM) using a shared library named `libnvram.so` to persist device-specific configuration parameters. For routers and other networking equipment, this includes settings shown on the web-based configuration interface, which can include wireless network settings, network adapter MAC addresses, and access credentials for the web interface.

Since this peripheral is typically abstracted as a key-value store, we developed a custom userspace library that intercepts calls to NVRAM-related functions, such as `const char *nvram_get(const char* key)` and `int nvram_set(const char* key, char *val)`, which are respectively used to get and set parameters from NVRAM. By modifying the system environment passed by the kernel to the `init` binary to include this library via `LD_PRELOAD`, we ensure that all userspace processes inherit the same environment, since they are child processes of `init`. A temporary mountpoint on the filesystem is used as the root of our key-value store, allowing us to reimplement this interface in userspace without emulating hardware-specific peripherals.

During testing, a common challenge we encountered was that our dataset of firmware images was compiled with different C toolchains, some of which we do not have access to. As shown in §V-A, this diversity was problematic for our shared library, since all dynamically-loaded ELF binaries must specify the path to the dynamic loader for which they were compiled, as well as the filenames of dynamically-loaded dependencies, which were different depending on the system.

Initially, we attempted to resolve this problem by compiling our NVRAM implementation statically. However, we soon discovered that not only did these C runtime libraries use incompatible implementations of built-in C features such as thread-local storage, but they were also not built as position-independent code (PIC) to support static compilation. As a result, we could neither build our NVRAM library statically against a single C runtime library, nor could we dynamically build our shared library specifically for each firmware image.

Fortunately, ELF dynamic loaders for Linux systems support lazy linking, which allows the resolution of external function symbols to be delayed until usage. Typically, the compiler implements this by placing stub code within the Procedure Linkage Table (PLT) that initializes the Global Offset Table (GOT) entry for a given imported function when the function is called for the first time.

Since the ELF loader uses a global symbol lookup scope during resolution [12], we were able to compile our NVRAM library with the `-nostdlib` compiler flag, delaying resolution of external symbols until after the *calling process* had already loaded the system C runtime library. Effectively, this allowed our shared library to appear as a static binary while dynamically utilizing functions made available by the calling process, including the standard C runtime library.

Another challenge we encountered was the fact that our NVRAM implementation was not useful without a set of system-specific default values. Unfortunately, these values are normally embedded within the hardware NVRAM peripheral at the factory, and having a hardware dependency for our system would preclude our goal of performing a large-scale analysis. Simply returning `NULL` or the empty string was also insufficient, as this would eventually cause the system to crash at startup or enter an erroneous state, e.g., by calling `itoa()` or `strcpy()` on a `NULL` pointer, or inserting bad arguments to program invocations such as `ifconfig`. Initially, we attempted to hardcode a set of default NVRAM values into our library, but we soon discovered that this was infeasible since an average firmware image can reference hundreds of NVRAM keys at startup.

After manually examining firmware images that failed to emulate, we realized that most images embedded a set of default NVRAM values into a few common locations, e.g., within a text file named `/etc/nvram.default`, `/etc/nvram.conf`, or `/var/etc/nvram.default`. Others would export a symbol `router_defaults` or `Nvrams` of type `char *[]` within built-in libraries such as `libnvram.so` or `libshared.so`. We were able to access these symbols by declaring them as weak references and checking if they were initialized, since we could not utilize `libdl.so` (not typically loaded by the calling executable) or leave them as regular references (external data symbol resolution is not lazy).

Unfortunately, our NVRAM emulation implementation does not work for all firmware images. This can be due to a wide variety of reasons. For example, some images may call NVRAM-related functions that we do not emulate; others may expect different semantics from these emulated functions in terms of parameter passing, return values, or caller/callee memory allocation; some others may implement NVRAM as a custom data structure on a MTD partition, which we currently cannot initialize to a valid state. We believe failures in NVRAM emulation are likely to be a significant contributor to the drop in emulation progress between columns two and three of Fig. 2. As an inconvenient truth, improving the emulation success rates or fixing network configuration detection for firmware images from, e.g., Tomato by Shibby, is a manual process. It requires an analyst to manually examine system logs in order to identify and classify emulation failures based on root cause, then make the changes that are necessary to support these images. Oftentimes, this may be a cyclic process, as there can be multiple causes of emulation failure.

2) Kernel: As mentioned in §II-B, we do not utilize the extracted kernel, but instead replace it with our own custom pre-built kernels for the ARM and MIPS architectures, which together account for 90.8% of our dataset.

During the kernel compilation process, we implement our analysis within our custom Linux kernel module that is used to aid debugging and emulating the original system environment. By hooking 20 system calls using the kernel dynamic probes (`kprobes`) framework, we are able to intercept calls that alter the execution environment. This includes operations such as assigning MAC addresses, creating a network bridge, rebooting the system, and executing a program, all of which are monitored by our framework to properly configure the emulated networking environment. This functionality can also be used to provide automatic confirmation of vulnerabilities, especially in conjunction with predefined poison values (e.g., `0xDEADBEEF`, `0x41414141`) that should never appear in system calls.

Since some firmware images expect certain filesystems to be mounted at boot, e.g., `/dev` or `/proc`, we use the `rdinit` kernel parameter to run a custom script that initializes these filesystems before `init` is executed. Additionally, we load the `nandsim` kernel module at startup, which emulates the memory technology device (MTD) partitions accessed via `/dev/mtdX` that are frequently used on these embedded devices.

In addition, since our emulation of NVRAM is volatile, we prohibit the guest from rebooting the system and emulate this behavior by restarting the `init` process. This kernel module also emulates vendor-specific or device-specific interfaces, such as custom device nodes, `procfs` entries, or non-standard `IOCTL`'s by returning success with a generic stub.

For the MIPS architecture, we build separate kernels for big-endian and little-endian systems, both targeting the MIPS Malta development platform, which is well-supported by both `QEMU` and the Linux kernel. In fact, this platform even supports MIPS 64-bit code, although we have not implemented support for it since it comprises less than 0.6% of our dataset. This kernel is currently at version 2.6.32.68, which is a long-term support release, and includes our backported commits for full `kprobes` support.

For the ARM architecture, we support only little-endian systems, since big-endian systems comprise less than 1.1% of our dataset and are unsupported by mainline `QEMU`[3]. We target the ARM Versatile Express development platform, which uses

[3] https://lists.gnu.org/archive/html/qemu-devel/2014-06/msg03257.html

an emulated Cortex-A9 (ARMv7-A) processor. This platform offers better hardware compatibility than the standard ARM Versatile Platform Baseboard development board, which uses an emulated ARM926 (ARMv5) processor that does not support newer ARM instructions found in some firmware images. Unfortunately, this platform supports only up to one emulated Ethernet device due to the lack of an emulated PCI bus in QEMU. In the future, we plan to switch to the ARM Virtual Machine platform, which supports multiple virtualized devices via VirtIO, but this will require a kernel upgrade from 3.10.92 to 4.1.12, a newer long-term support release that fully supports VirtIO functionality on ARM.

As the above discussion suggests, adding support for a new hardware architecture, such as x86, is not an automated process. In particular, selecting a supported hardware platform in QEMU can be tricky, as support for either VirtIO or an emulated PCI bus is typically required to attach more than one virtual networking interface. At the same time, the chosen hardware platform in QEMU must be supported by the selected version of the Linux kernel, which needs to be sufficiently up-to-date for `kprobes` and VirtIO support. Developing a compatible configuration for the kernel can also be tricky, as we need to enable all the features that off-the-shelf firmware relies on. Furthermore, we need to rebase our custom kernel module implementation to the chosen kernel version, which may require manual compatibility fixes to account for internal kernel API changes.

3) System Configuration: Since we are mainly interested in firmware that implements network functionality, such as routers, network attached storage, or surveillance equipment, we need to make device-specific changes to the emulated hardware. Ideally, all network devices would automatically configure themselves via the DHCP protocol. Unfortunately, certain network devices, especially routers and some managed switches, are designed to provide DHCP services to other devices. Additionally, these devices tend to have different numbers of network interfaces; for example typical consumer routers have at least four Ethernet interfaces, in comparison to just one on an IP camera.

Our system initially executes each emulated firmware in a "learning" phase for 60 seconds. In this phase, the emulator is configured with the default hardware peripherals for the emulated target platform (MIPS Malta or ARM Virtual Express), plus up to four emulated network adapters, using the built-in socket networking backend within `QEMU`. During this time, information is gathered about the expected network configuration. In particular, we keep track of IP addresses that are assigned to network interfaces, as well as the presence of IEEE 802.1d bridges used to aggregate multiple network interfaces. Additionally, we check for tagging and separation of Ethernet frames using IEEE 802.1Q VLANs, which is used by some routers to segregate wireless guest networks from the physical network.

This information is then fed back into our emulation framework to develop a more accurate `QEMU` configuration for this system. We instantiate a network tap (TAP) device on the host, which is associated with one of the emulated network interfaces within the firmware (e.g., `eth0`) that correspond to a LAN interface. For firmware images that use VLANs, we assign a corresponding VLAN ID to the TAP interface, in order to communicate successfully with emulated network services. Next, the `TAP` interface is configured with an IP address that resides in the same subnet as the IP address assigned to the emulated interface by the firmware. Finally, we check for network connectivity by sending ICMP requests and performing a port scan using the `Nmap` [3] utility.

4) QEMU: Aside from NVRAM, we expect embedded systems to rely on other hardware-specific peripherals such as watchdog timers or additional flash storage devices. Unfortunately, some device manufacturers do not follow good software engineering practices and implement such functionality directly in userspace, instead of using a device driver in kernelspace.

As a result, we cannot simply abstract away these devices and cleanly emulate this behavior within our custom kernel module. For example, the `alphafs` webserver mentioned in §II-B maps part of physical memory from the `/dev/mem` device node directly into its own address space. It expects configuration information for the flash memory chip to be mapped at `0x1e000000`, with the `VendorID` and `ProductID` identification parameters matching a chip supported by the software; otherwise it simply terminates.

To support the 138 affected firmware images in FIRMADYNE, we modified the appropriate sixteen bytes in `QEMU`'s source code for the emulated platform flash device to respond with known good values.

D. Automated Analyses

Currently, we have implemented three basic automated dynamic analysis passes within our system. Each is registered as a callback within our system, such that when a firmware image enters the network inferred state, registered callbacks are triggered sequentially. These contributed to our detection of 14 previously-unknown vulnerabilities that affect 69 firmware images, and 74 known vulnerabilities that affect 887 firmware images (see §V).

1) Accessible Webpages: To help detect various information disclosure, buffer overflow, and command injection vulnerabilities, we wrote a simple analysis that looks for publicly accessible webpages from the LAN interface of firmware images. A custom-written Python test harness iterates through each file within the firmware image that appears to be served by a webserver (e.g., located within `/www/`), verifies that it is not a static resource (e.g., *.png, *.css, *.js), and attempts to access it directly over the web interface.

Responses that contained non-2xx HTTP status codes were ignored, since these were typically inaccessible web pages (403/404), web pages that required authentication (401), or invalid responses caused by socket timeouts or incomplete reads. Successful responses that contained redirects were flagged as lower confidence results, since we experimentally determined that a large number of these were used to implement soft-authentication pages.

Perhaps as a more user-friendly authentication mechanism, these soft-authentication pages checked whether client requests were authenticated using a client cookie or server IP address log instead of the basic or digest authentication mechanisms

built-into the HTTP protocol (which would return 401). Thus, these pages were marked with lower confidence, while all other web pages were marked with regular confidence. These results were aggregated across our firmware dataset to determine which URLs were most accessible, and then prioritized for further analysis in order of popularity.

2) SNMP Information: We were curious about the prevalence and security of Simple Network Management Protocol (SNMP) implementations across our dataset, and so we wrote a basic analysis using our framework to dump all unauthenticated SNMP information from the "public" and "private" communities using the `snmpwalk` tool. Using MIB files gathered by the crawler, the results for a subset of these were manually interpreted to check for the presence of sensitive information. The corresponding object identifiers (OIDs) were recorded, and a simple proof-of-concept was developed for each, based on whether information was returned when the OID was queried.

3) Vulnerabilities: Using 60 known exploits, mostly from the Metasploit Framework, we initially checked all firmware images across our dataset for known security vulnerabilities. Each exploit was executed sequentially, with a remote shell payload if applicable, then the corresponding exploit log was checked for success. This provided a lower-bound on the number of vulnerabilities within our dataset, since an exploit may fail even if a vulnerability is present. The tested vulnerabilities were manually selected for relevance to applications and daemons known to be present on embedded devices, and spanned various exploit categories such as buffer overflow, command injection, information disclosure, and denial of service.

For the new vulnerabilities that we discovered, we manually developed proof-of-concepts exploits, which leveraged our predefined poisoned arguments such as `0xDEADBEEF`. Then, we specified a verification condition for each exploit, which was typically the presence of the poisoned argument in our instrumented kernel log; other examples included a segmentation fault at `0x41414141` or a WPS PIN in a webpage.

E. Additional Capabilities

We also developed a number of additional capabilities that assisted the development and debugging of our emulation framework and exploits. These include dynamic tracing of code execution, which can be imported into existing reverse engineering tools, such as *IDA Pro*. Our custom kernel was modified to disable inlining of the `context_switch()` function, which allowed the emulator to trace the execution of given userspace processes. Additionally, at startup we also launch a special console application on the device node `/dev/ttyS1`, which is forwarded by `QEMU` to a temporary socket on the host system. This provided us with a convenient mechanism for modifying the emulated firmware image at runtime, especially if no default console is launched.

V. Evaluation

In this section, we evaluate our implementation of FIRMADYNE. First, we examine the composition of our input dataset, and analyze its effect on the emulation fidelity at every stage in the emulation pipeline. Second, we demonstrate how we leveraged our system to identify 14 previously-unknown vulnerabilities within the collected firmware samples. Using proof-of-concept exploits that we developed for each of these vulnerabilities, we use our system to assess their prevalence and impact on our dataset. Finally, we demonstrate the analysis flexibility of our system by supplementing it with 60 known exploits, mostly from the Metasploit Framework [2], and assess the prevalence and impact of these known exploits on our dataset.

It is important to note that the distribution of firmware images across product lines and device vendors is not uniform, and thus may skew interpretation of the results. In particular, although we attempt to scrape metadata about the model number and version number of each firmware image, this information is not always available, nor is it present in a format that can easily establish a temporal ordering. For example, vendors may re-release a given product with different hardware, or release a product with different hardware or firmware in each region, preventing direct comparisons between two firmware images with the exact same model. As a result, it is difficult to identify which firmware images are deprecated, and which firmware image(s) is (are) the current version(s).

Furthermore, it is difficult to establish a mapping between firmware images and products, since there is not a direct one-to-one correspondence. For example, some vendors, such as Mikrotik, distribute a single firmware image for each hardware architecture whereas other vendors, such as OpenWRT, distribute a single firmware image for each hardware chipset. Additionally, some vendors, such as QNAP and Synology, develop a master firmware image that is only lightly customized for each product in terms of hardware support and product strings, whereas other vendors, such as OpenWRT, distribute different binary releases of the same firmware image using various encapsulation formats. Given two different firmware binaries, this raises the question of how functionally identical they may be, which we do not address. Nevertheless, we attempt to provide a lower-bound on the number of affected products, where possible.

A. Statistics

1) Architectures: For all firmware images with extracted root filesystems, we were able to identify the architecture of the corresponding firmware image by examining the format header of the `busybox` binary on the system, or alternatively binaries in `/sbin/` if we could not locate `busybox`.

Table I shows that the majority of our firmware images are 32-bit MIPS (both big-endian and little-endian), which constitute approximately 79.4%. The next most popular architecture type is 32-bit little-endian ARM, which constitutes approximately 8.9%. Combined, these two architectures constitute 90.8% of all firmware images, with the remainder forming the little-tail of this distribution, suggesting that additional development effort to support the remaining architectures would require some other strong justifications.

2) Operating Systems: By combining our statistics for root filesystem extraction and signature matches for the Linux and VxWorks kernels, we noticed that the largest proportion of our firmware images were UNIX-based at 48%, as shown in Table II. If the filesystem of a firmware image was positively

identified as UNIX-based, but failures were encountered during the kernel extraction process, then the image was labeled as UNIX-like. Potential causes for this include path exploration constraints, unsupported compression algorithms, or even the lack of a kernel within the firmware image. Barely 3.5% of our firmware images were identified as VxWorks, showing that implementing support for these devices is a low priority.

As discussed previously in the last paragraph of §IV-B, the unknown firmware images can be attributed to a number of extraction failures. These include firmware images that appeared to be Linux-based, but for which we were unable to reassemble the entire filesystem, extracted only a partial UNIX-like filesystem, or extracted a filesystem that did not meet our threshold to be deemed UNIX-like. Some of these are known to use ZynOS, a proprietary real-time operating system developed by ZyXEL Communications. ZynOS uses the ThreadX kernel and an unknown filesystem type, for which we lack a kernel version signature and filesystem extraction utility.

Other unknown firmware images are monolithic firmware images that do not utilize a distinct kernel or filesystem. As a result, emulating these firmware images would be extremely difficult without hardware documentation, as chipset-specific code may be distributed throughout the binary. This type of firmware image is known to be used by u-blox, which is included in our dataset.

3) Kernel Modules: Across all of our extracted firmware images, we performed a basic categorization of all out-of-tree kernel modules based on filename, shown in Table III. These numbers indicate that 58.8% of these modules implement various network-related functionality, such as packet filtering (`iptables`, `xtables`, `netfilter`, `ebtables`), protocol implementations (`pptp`, `ppp`, `adsl`), and interface support (`mii`, `tun`, `tap`). The next largest subset of 12.7% were used to provide support for various peripherals, including wireless adapters (`wl`, `ath9k`, `sierra`), platform chipsets (`ar7240`, `ar7100`, `bcm963xx`), and various other devices (`acos_nat`, `pl2303`). Many of the remaining kernel modules appeared to be in-tree kernel modules that were compiled as loadable modules, including generic USB interface implementations (`ehci`, `uhci`, `xhci`), filesystems (`fat`, `fuse`, `ext3`), cryptographic functions (`sha512`, `crypto`), and various other miscellaneous kernel routines (`ts_fsm`, `sch_hfsc`). Less than 0.2% of these kernel modules were identified as the KCodes NetUSB kernel module, a proprietary USB over IP kernel module that is known to contain a remotely-exploitable buffer overflow vulnerability.[4]

4) Network Services: To assess the prevalence of listening network services on our firmware image dataset, we used the `nmap` network scanning tool to check the 1,971 images that respond to ICMP echo requests. We scanned all TCP ports with known services from the `nmap-services` file, as well as the continuous port range 1–1024, which is the default scanning behavior of `nmap`. The top ten results, shown in Table IV, indicate that out of the 1,971 devices that were network reachable, 47.3% likely support a web-based configuration

[4]https://www.sec-consult.com/fxdata/seccons/prod/temedia/advisories_txt/20150519-0_KCodes_NetUSB_Kernel_Stack_Buffer_Overflow_v10.txt

Architecture (Endian)	# Image(s)
TILE (LE)	1
ARC (LE)	10
Motorola 68k (BE)	10
x86 (LE)	31
MIPS 64-bit (BE)	50
PPC (BE)	84
ARM (BE)	102
x86-64 (LE)	147
Unknown	439
ARM (LE)	843
MIPS (BE)	3,137
MIPS (LE)	4,632
Total	9,486

TABLE I: Breakdown of firmware images by architecture, based on binary fingerprinting of extracted root filesystems.

Type	# Images
Linux	9,379
Unidentified (UNIX-like)	2,187
VxWorks	857
Unknown	10,612
Total	23,035

TABLE II: Breakdown of firmware images by operating system, based on kernel fingerprinting and root filesystem extraction.

Category	# Modules
NetUSB	853
Unclassified	1,384
Cryptography	12,603
USB	30,683
Filesystems	43,271
Miscellaneous	55,344
Peripheral Drivers	64,085
Networking	296,592
Total	504,815

TABLE III: Breakdown of kernel modules by category, based on path and filename.

# Images	TCP Port/Service	# Vendor(s)
928	80/http	9
708	23/telnet	7
536	53/domain	6
250	3333/dec-notes	1
188	443/https	7
187	5000/upnp	2
136	1900/upnp	1
162	49152/unknown	4
63	2602/ripd	2
57	5555/freeciv	3

TABLE IV: Breakdown of listening network services by number of firmware images and number of vendors.

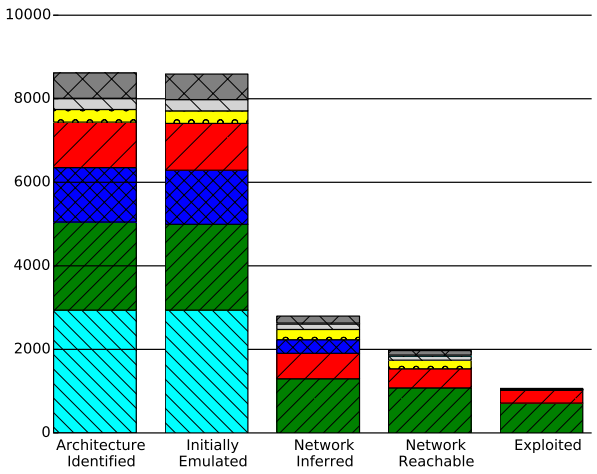

Fig. 2: Breakdown of firmware images by emulation progress, colored by vendor.

interface (HTTP or HTTPS). Of these, only 9.5% supported `HTTPS` for the configuration interface, which is 19.8% of the devices that support `HTTP`.

Remote shell access is supported by 37.4% of devices over either the `Telnet` or `SSH` protocols. Note, however, that `SSH` is not among the top ten results; in fact, it ranks 13th at 2.2%, or 1.9% of the devices that support `Telnet`. This is worse than the percentage of devices supporting `HTTPS` out of the devices that support `HTTP`.

Based on the presence of the `DNS` service, it appears that at least 27.2% of these firmware images are routers, which typically act as a local DNS proxy. Another 16.4% ship with Universal Plug and Play (UPnP) enabled by default, which allows LAN devices to automatically configure port forwarding from the WAN interface.

Port 2602 is known to be used by the Routing Information Protocol (RIP) protocol, which is typically enabled on enterprise-class routers for automatic network routing.

According to various customer support forums, ports 3333 and 5555 are known to be open on certain Netgear devices, although we have not checked our Netgear firmware images to identify the responsible service. Port 49152 is known to be the first port in the dynamic port address range forwarded by various applications through UPnP, though we do not have any UPnP clients in our network configuration and are uncertain of the default forwarding state.

5) Emulation Progress: As shown in Fig. 2, of the 8,617 extracted firmware images for which we identified an architecture, our system initially emulated 96.6% (8,591) successfully. The failures can be attributed to a number of causes, including the lack of an `init` binary in a standard location (`/bin/init`, `/etc/init`, or `/sbin/init`), or an unbootable filesystem. For example, certain images containing Ralink chipsets are known to name their `init` binary `ralink_init`, which we currently do not support. Likewise, extraction failures discussed in the last paragraph of §IV-B can also affect success of the initial emulation. Since we only extract the first UNIX-like filesystem from firmware images that contain multiple filesystems, it is likely that only part of the filesystem has been extracted, leading to a boot failure. Reassembling such systems into a single filesystem is not straightforward because each filesystem can potentially be mounted on top of another at arbitrary locations.

Of the 8,591 firmware images that entered the "learning" phase, only 32.3% (2,797) had their networking configuration successfully inferred. We believe that this decrease occurred due to failures in the boot process while attempting to infer the network configuration. As we previously discussed in the last paragraph of §IV-C1, problems with NVRAM emulation are a significant contributor to these failures. For example, some routers may not initialize correctly if our NVRAM implementation was not able to override the built-in implementation, if insufficient default NVRAM values were loaded by our implementation, or if the built-in NVRAM implementation expected different semantics from NVRAM-related functions. These manifest as various crashes or hangs during the boot process, especially if memory or string manipulation functions (`memcpy()`, `strcpy()`, etc.) are called on `NULL` values returned by our NVRAM implementation for nonexistent keys. Additionally, it is also possible that some images do not use a NVRAM hardware peripheral, but instead write configuration values directly to a MTD partition, which we may not successfully emulate.

Other potential sources of networking failures include different naming conventions for networking devices. For example, devices that utilize Atheros or Ralink chipsets may expect platform networking devices to be named similarly to `ath0` or `ra0`, respectively, instead of the generic `eth0`. Likewise, other devices may expect the presence of a wireless networking interface such as `wlan0`, and fail otherwise. In addition, since our ARM little-endian emulation platform currently supports only up to one emulated Ethernet device, this may prevent some firmware images from correctly configuring networking.

Only 70.8% (1,971) of the 2,797 images with an inferred network configuration are actually reachable from the network using `ping`. This may be caused by firewall rules on the emulated guest that filter ICMP echo requests, resulting in false negatives, or various other network configuration issues. For example, our system may have mistakenly assigned the host TAP interface in `QEMU` to the WAN interface of the emulated device instead of a LAN interface, or identified the default IP address of the WAN interface instead of the LAN interface. Similarly, firmware may change the MAC address of the emulated network device after it has booted, resulting in stale ARP cache entries and a machine that appears unreachable.

Surprisingly, our results show that 45% (887 out of 1,971 firmware images) of the network reachable firmware images are vulnerable to at least one exploit. We discuss this further in §V-B, where we give a breakdown by exploit.

Exploit ID	# Images	# Products	Affected Vendor(s)
47	282	16	21, 22, 37
56	169	14	16, 21, 35
64	169	27	12, 21, 37
45	136	13	21
43	88	10	12
202	49	11	12, 16, 21, 36, 37, 42
207	35	6	21
60	31	9	7, 12, 19, 21, 37
205	16	5	21
206	14	4	21
203	13	5	12
59	9	N/A	12
200	8	1	21
201	7	1	21
210	7	2	12
4	6	N/A	12
24	5	1	19, 42
213	4	1	21
214	4	1	21
39	3	N/A	12
209	3	1	12
212	3	1	21
61	2	1	42
204	1	N/A	21
211	1	1	21

TABLE V: Breakdown of exploits by number of affected firmware images, number of affected products, and affected vendor(s), indexed into Table VII. Note: `N/A` indicates that we do not have sufficient metadata to compute a lower-bound on affected products.

# Exploits	# Images	# Vendor(s)	# Products
5	2	1	1
4	8	1	3
3	30	2	10
2	86	5	14
1	761	9	77
0	1,910	22	263
Total	2,797	42	322

TABLE VI: Breakdown of successful exploits by number of firmware images, number of vendor(s), and number of affected products.

B. Results

In Table V, we provide a breakdown of all successful exploits by exploit. Exploits in the range #0–#100 are sourced from the Metasploit Framework, whereas most exploits greater than or equal to #200 are previously-unknown vulnerabilities for which we developed proof-of-concepts (POC's). This excludes #202, which is a known vulnerability but not sourced from the Metasploit Framework. Each of these previously-unknown vulnerabilities has been reported to the respective vendor, following the policies of responsible disclosure. We discuss a few specific vulnerabilities below in greater detail as case studies.

By tabulating the results from Table V for each firmware image, we obtain Table VI, which provides a breakdown of the firmware images by the number of successful exploits. This shows that a small number of these firmware images are vulnerable to more than two exploits, with the least secure image suffering from five exploits. Interestingly, all 40 of these firmware images vulnerable to more than two exploits are routers and access points manufactured by D-Link and Netgear; however, this data may be skewed by the distribution of our exploits and firmware images, which is not uniform. These results initially seem to decay exponentially, with less than half (39.8%) of firmware images vulnerable to zero exploits being vulnerable to one exploit, but then there is a long-tail in the vulnerability distribution, with only 4.5% (126) of firmware images affected by more than one exploit.

1) Command Injection (#200, #201, #204–#206, #208): While analyzing the aggregate results of our automated accessible webpages analysis (§IV-D1), we discovered six previously-unpublished command injection vulnerabilities that affect 24 firmware images for wireless routers and access points manufactured by Netgear. All six vulnerabilities were within PHP server-side scripts that provided debugging functionality but appeared to be accidentally included within production firmware releases. In particular, five of these were used to change system parameters such as the MAC address of the WLAN adapter, and the region of the firmware image (e.g., World Wide [WW], United States [US], or Japan [JP]). The remaining one was used to write manufacturing data such as MAC address, serial number, or hardware version into flash memory. Our manual analysis of the PHP source code revealed that all were straightforward command injection vulnerabilities through the `$_REQUEST` super-global and unsafe use of the `exec()` function. After discovering these potential vulnerabilities, we leveraged FIRMADYNE to automatically verify their exploitability across our entire dataset.

2) Buffer Overflow (#203): Another new vulnerability that we manually discovered, using the results of our automated accessible webpages analysis, was a buffer overflow vulnerability within firmware images for certain D-Link routers. To implement user authentication, the webserver sets a client-side cookie labeled `dlink_uid` to a unique value that is associated with each authenticated user. Instead of verifying the value of this cookie within the server-side scripting language of the webpage, this authentication functionality was actually hard-coded within the webserver, which uses the standard library functions `strstr()`, `strlen()`, and `memcpy()` to copy the value of the cookie. As a result, we were able to set the value of this cookie to an overly-long value to cause the webserver to crash at `0x41414141`, another poisoned argument that we monitor for.

3) Information Disclosure (#207, #209–#214): Using the automated webpage analysis, we also discovered seven new information disclosure vulnerabilities across our dataset that affect 51 firmware images for various routers manufactured by both D-Link and Netgear. One of these (#207) was within an unprotected webpage that provides diagnostic information for the router, including the WPS PIN and passphrases for all locally-broadcast wireless networks.

The remaining six vulnerabilities (#209 – #214) were within the Simple Network Management Protocol (SNMP) daemon of both manufacturers. This feature was enabled by default likely because these routers were targeted towards small businesses rather than home users. To interpret results obtained from SNMP queries, one needs access to a Management Information Base (MIB) file that describes the semantics of each individual object (OID) field. As discussed in §III-B, our crawlers record links to MIB files in the collected metadata, enabling manual verification of the obtained results.

Our automated exploit verification showed that these firmware images would respond to unauthenticated SNMP v2c queries for the `public` and `private` communities, and return values for the OID's that contain web-based access credentials for all users on the device, and wireless credentials for all locally-broadcast wireless networks.

4) Sercomm Configuration Dump (#47): This exploit, reported as CVE-2014-0659[5] and sourced from the Metasploit Framework, attacks undocumented and badly-designed features of the `scfgmgr` service to remotely dump system configuration variables from NVRAM and obtain a shell. Public documentation for this vulnerability suggests that, as of 2015-01-28, it was known to affect firmware for networking devices manufactured by Cisco, Linksys, Netgear, and a variety of smaller vendors. This is corroborated by our automated analysis, which also confirmed the presence of this vulnerability within devices manufactured by On Networks and TRENDnet. More precisely, our results suggest that this single vulnerability affects 14.3% of all network reachable firmware images from our dataset. This is because Sercomm Corporation is likely the original equipment manufacturer (OEM) for these devices, which were then re-branded and re-sold by various vendors.

5) MiniUPnPd Denial of Service (#56): Reported as CVE-2013-0229[6], this exploit takes advantage of parsing flaws for the Simple Service Discovery Protocol (SSDP) within MiniUPnP[7], an open-source UPnP daemon implementation, to trigger a denial of service attack on this service.

According to our results, 8.5% of all network reachable firmware images from our dataset are vulnerable to this attack, which was fixed on 2009-10-30 with the release of MiniUPnP 1.4. Affected vendors include Huawei, Netgear, and Tomato by Shibby, which is a community-developed third-party firmware for various wireless routers. Statistics released by Rapid7, the developers of the Metasploit Framework and the original reporters of this vulnerability, indicate that as of 2013-01-29, 332 products used MiniUPnP 1.0, with over 69% of all MiniUPnP fingerprints corresponding to version 1.0 or older. Again, these results emphasize the prevalence of cross-vendor vulnerabilities due to shared software components, whether open-source or proprietary.

6) OpenSSL ChangeCipherSpec (#64): This vulnerability was reported as CVE-2014-0224[8], and takes advantage of a bad state machine implementation for the SSL/TLS handshake process in all versions of OpenSSL before 0.9.8za, 1.0.0m, and 1.0.1h. Exploitation of this vulnerability allows an attacker to downgrade the cipher specification between a client and a server, potentially permitting a man-in-the-middle (MITM) attack. Our results show that 8.5% of all network reachable firmware images are vulnerable to this attack, which is 89.9% of all firmware images that accept `HTTPS` connections. This exploit also affects 8.4% of all products in our dataset, the most out of all exploits. Affected vendors include D-Link, Netgear, and TRENDnet.

[5] https://github.com/elvanderb/TCP-32764

[6] https://community.rapid7.com/servlet/JiveServlet/download/2150-1-16596/SecurityFlawsUPnP.pdf

[7] http://miniupnp.free.fr/

[8] http://ccsinjection.lepidum.co.jp/

C. Discussion and Limitations

Although FIRMADYNE performed well in our experiments, there is certainly room for improvement. As discussed previously in §IV-B, §IV-C1, and §IV-C2, additional manual effort can improve the system by, e.g., fixing extraction failures, adding support for additional hardware architectures, or correcting emulation failures. These changes require an analyst to manually classify failures by root cause and perform the changes that are necessary to increase compatibility. Implementing a new analysis pass also requires manual labor, though we can potentially reap a large benefit from it because each newly-implemented analysis can be automatically executed on all supported firmware images from our dataset.

In addition, as mentioned in §V, our results can be difficult to evaluate due to the lack of a mechanism for quantifying real-world impact in terms of unique products (instead of unique firmware images). Likewise, our results are affected by skew caused by differences in vendor composition of our dataset, and of network reachable firmware images.

Other limitations of FIRMADYNE include the usage of custom pre-built kernels, which currently do not load out-of-tree kernel modules from the filesystem. As a result, our system cannot be used to confirm vulnerabilities in kernels or kernel modules shipped by the vendor within firmware images. For example, we are unable to assess the prevalence of the KCodes NetUSB kernel module buffer-overflow across our dataset because of this limitation.

Likewise, we do not identify which network port is used as the uplink (or WAN) port, and which network port(s) are used for the downlink (or LAN) port(s). This prevents us from determining whether detected vulnerabilities are exploitable from the Internet, or only by locally-connected clients.

Nevertheless, a number of techniques can be used by remote attackers to pivot from the WAN interface to the LAN interface over a web browser, including Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS), or even DNS rebinding attacks. Additionally, with the increasing deployment of IPv6, local machines are now being assigned globally-routable IP addresses. This potentially allows attackers to access the LAN interface of consumer devices, even though routers can still act as firewalls. An increasing number of wireless routers and access points also now support network isolation or client isolation features, which can segregate traffic between various wireless or physical interfaces. However, the presence of these vulnerabilities within the gateway router clearly compromises this protection.

VI. Related Work

With the increasing prevalence of embedded devices, several related works have performed large-scale analyses of firmware images, using a variety of analysis techniques. For example, Heffner[9] performed large-scale extraction of embedded firmware images to gather a database of over $2,000$ hardcoded SSL private keys. Likewise, Rapid7[10] used a similar analysis for hardcoded SSH private keys, albeit on a smaller scale.

Using static analysis, Costin et al. [8] recently analyzed a dataset of approximately $32,000$ firmware images. They discovered a total of 38 previously-unknown vulnerabilities, including hard-coded back-doors, embedded private key-pairs, and XSS vulnerabilities, all of which were obtained "without performing sophisticated static analysis".

Another effective technique for large-scale measurement of embedded device security is network scanning, which avoids direct analysis of firmware images. Using tools such as Nmap, Cui and Stolfo [10] identified approximately $540,000$ publicly-accessible embedded devices with *default* access credentials. Over the course of a 4-month longitudinal study, they discovered that less than 3% of access credentials were changed, which suggests that user awareness is lacking. Likewise, using the ZMap [13] network scanner, Heninger et al. [14] showed that embedded devices can also suffer from entropy problems. Their results indicate that 2.45% of TLS certificates may be vulnerable to brute-force attacks due to faulty RSA key generation, and that 1.03% of DSA private keys are factorable due to nontrivial common factors.

Additionally, previous work has discovered specific vulnerabilities that affect various classes of embedded devices. Using HP LaserJet printers as a case study, Cui et al. [9] demonstrated that remote firmware update functionality can be exploited by attackers to insert malware. Weinmann [18] showed that deployed cellular baseband implementations suffer from remotely exploitable memory corruption vulnerabilities, which can be used to execute arbitrary code on the baseband processor. Similarly, Bonkoski et al. [6] showed that remote management functionality on server motherboards is riddled with security vulnerabilities, allowing a remote attacker to take control of the system. Finally, Maskiewicz et al. [16] and Nohl et al. [17] showed that malicious functionality can be inserted into the firmware of USB peripherals, allowing an attacker to take control of host systems and exfiltrate data.

To defend against this attack vector, several different techniques have been developed to find vulnerabilities in embedded devices. For example, Davidson et al. [11] have developed a symbolic executor using the KLEE [7] symbolic execution engine to detect vulnerabilities in embedded devices. Their work discovered 21 memory safety bugs across a corpus of 99 open-source firmware programs for the MSP430 family of 8-bit embedded micro-controllers. At a lower level, Li et al. [15] ported the `QEMU` emulator into the BIOS to model hardware peripherals for validation of an embedded SoC during development.

Recently, Zaddach et al. [19] have also developed a framework for performing dynamic analysis of embedded firmware by forwarding I/O accesses from within an emulator the actual hardware for execution. However, this approach does not scale in terms of analysis cost and time, which is why we have designed FIRMADYNE to perform robust hardware emulation and vulnerability verification in an *automatic* manner.

VII. Conclusion & Future Work

By developing FIRMADYNE, our automated dynamic analysis framework, we hope to lower the bar for discovering new vulnerabilities within embedded systems. At the same time, FIRMADYNE implements an automated approach to assess the prevalence of newly-discovered security vulnerabilities in a large population of embedded device firmware images. Given the weak security posture of these devices, we believe that greater attention to these devices by security researchers, hobbyists, and other interested parties can motivate device manufacturers to address security issues in their products more swiftly. This is especially true for OEMs, who are responsible for a significant fraction of the vulnerabilities in existing deployed devices.

As shown in Fig. V-A, the next-largest category (after Linux) of embedded firmware from our dataset are from various proprietary real-time operating systems (RTOS) such as VxWorks. This presents a potential avenue for future work, especially given the existence of published vulnerabilities that affect these platforms. In particular, we would be interested in developing a compatibility layer for these applications using existing real-time Linux development frameworks such as Xenomai on our emulation platform.

A considerable number of source code releases are available for many Linux-based embedded firmware due to the terms of common open-source software licenses. Since our dataset includes links to applicable source code for each firmware image, this could provide a mechanism for implementing effective static analysis, in conjunction with our existing framework for performing dynamic analysis.

Finally, statistical analysis techniques could be utilized to improve the firmware extraction component of our framework. Firmware images that appear obfuscated or encrypted could be handled by a separate extraction pathway. For example, it is well-known that firmware for Buffalo LinkStation devices are encrypted, but passwords and decryption utilities are publicly available.[11] The same applies to various firmware distributed for QNAP devices.[12]

Acknowledgment: This work was supported in part by grants from the Department of Defense through the National Defense Science & Engineering Graduate Fellowship Program and under contract no. N66001-13-2-4040, and the Office of Naval Research under grant N00014-15-1-2948. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

[9] https://github.com/devttys0/littleblackbox
[10] https://github.com/rapid7/ssh-badkeys
[11] http://buffalo.nas-central.org/wiki/Firmware_update
[12] http://pastebin.com/KHbX85nG

References

[1] "Binwalk." [Online]. Available: http://binwalk.org/

[2] "Metasploit." [Online]. Available: http://www.metasploit.com/

[3] "Nmap security scanner." [Online]. Available: https://nmap.org/

[4] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the USENIX 2005 Annual Technical Conference*. USENIX, 2005, pp. 41–46. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/bellard.html

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1646353.1646374

[6] A. Bonkoski, R. Bielawski, and J. A. Halderman, "Illuminating the security issues surrounding lights-out server management," in *Proceedings of the 7th USENIX Workshop on Offensive Technologies*. USENIX, 2013, pp. 1–9. [Online]. Available: https://www.usenix.org/conference/woot13/workshop-program/presentation/bonkoski

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*. USENIX, 2008, pp. 209–224. [Online]. Available: https://www.usenix.org/legacy/events/osdi08/tech/

[8] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium*. USENIX, 2014, pp. 95–110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin

[9] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium*. The Internet Society, 2013. [Online]. Available: http://www.internetsociety.org/doc/when-firmware-modifications-attack-case-study-embedded-exploitation

[10] A. Cui and S. J. Stolfo, "A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan," in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 97–106. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-78751540482&partnerID=40&md5=759904ebe0eca35e4297072f7224cf55

[11] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22nd USENIX Security Symposium*. USENIX, 2013, pp. 463–478. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson

[12] U. Drepper, "How to write shared libraries," 2006.

[13] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast internet-wide scanning and its security applications," in *Proceedings of the 22nd USENIX Security Symposium*. USENIX, 2013, pp. 605–619. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric

[14] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices," in *Proceedings of the 21st USENIX Security Symposium*. USENIX, 2012, pp. 205–220. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/heninger

[15] H. Li, D. Tong, K. Huang, and X. Cheng, "FEMU: A firmware-based emulation framework for SoC verification," in *Proceedings of the 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, no. 257. IEEE, 2010, pp. 257–266. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5751510&tag=1

[16] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: Exploiting firmware updates in USB peripherals," in *Proceedings of the 8th USENIX Workshop on Offensive Technologies*. USENIX, 2014, pp. 1–10. [Online]. Available: https://www.usenix.org/conference/woot14/workshop-program/presentation/maskiewicz

[17] K. Nohl and J. Lell, "BadUSB—on accessories that turn evil," 2014. [Online]. Available: https://www.blackhat.com/us-14/briefings.html#badusb-on-accessories-that-turn-evil

[18] R.-P. Weinmann, "Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks," in *Proceedings of the 6th USENIX Workshop on Offensive Technologies*. USENIX, 2012, pp. 1–10. [Online]. Available: https://www.usenix.org/conference/woot12/workshop-program/presentation/weinmann

[19] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proceedings of the 2014 Network and Distributed System Security Symposium*. The Internet Society, 2014, pp. 23–26. [Online]. Available: http://dx.doi.org/10.14722/ndss.2014.23229

Appendix

A. Dataset Breakdown

In Table VII to follow, we show the progress of FIRMADYNE in analyzing the firmware images in our dataset, grouped by vendor. Approximately 10% of all extracted firmware images were exploited.

Index	Vendor	Download	Extracted	Arch. Identified	Initial Emulation	Network Inferred	Network Reachable	Exploited
1	Actiontec	14 (6)	8 (4)	5 (3)	8 (4)	0 (0)	0 (0)	0 (0)
2	Airlink101	15 (12)	1 (1)	1 (1)	1 (1)	1 (1)	0 (0)	0 (0)
3	Apple	9 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
4	Asus	3 (1)	1 (1)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)
5	AT&T	25 (1)	6 (1)	4 (1)	6 (1)	2 (1)	0 (0)	0 (0)
6	AVM	132 (N/A)	7 (N/A)	7 (N/A)	7 (N/A)	0 (0)	0 (0)	0 (0)
7	Belkin	140 (61)	55 (29)	55 (29)	53 (29)	7 (4)	3 (2)	2 (2)
8	Buffalo	143 (61)	6 (5)	5 (4)	6 (5)	4 (3)	0 (0)	0 (0)
9	CenturyLink	31 (4)	9 (4)	9 (4)	9 (4)	1 (1)	1 (1)	0 (0)
10	Cerowrt	14 (N/A)	14 (N/A)	14 (N/A)	8 (N/A)	8 (N/A)	0 (0)	0 (0)
11	Cisco	61 (N/A)	43 (N/A)	39 (N/A)	34 (N/A)	2 (N/A)	0 (0)	0 (0)
12	D-Link	4,688 (434)	1,124 (113)	1,089 (109)	1,121 (119)	609 (65)	458 (48)	219 (32)
13	Forceware	2 (N/A)	2 (N/A)	2 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)
14	Foscam	56 (23)	5 (5)	5 (5)	5 (5)	5 (5)	0 (0)	0 (0)
15	Haxorware	7 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
16	Huawei	29 (17)	5 (3)	5 (3)	5 (3)	3 (2)	2 (1)	2 (1)
17	Inmarsat	47 (N/A)	2 (N/A)	2 (N/A)	2 (N/A)	2 (N/A)	0 (0)	0 (0)
18	Iridium	17 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
19	Linksys	126 (29)	105 (24)	101 (21)	105 (24)	43 (9)	36 (8)	5 (3)
20	MikroTik	13 (4)	5 (N/A)	4 (N/A)	2 (N/A)	0 (0)	0 (0)	0 (0)
21	Netgear	5,280 (372)	2,135 (156)	2,109 (155)	2,054 (149)	1,297 (92)	1,078 (79)	628 (47)
22	On Networks	28 (N/A)	15 (N/A)	15 (N/A)	15 (N/A)	11 (N/A)	10 (N/A)	7 (N/A)
23	Open Wireless	1 (N/A)	1 (N/A)	1 (N/A)	1 (N/A)	1 (N/A)	0 (0)	0 (0)
24	OpenWrt	1,498 (41)	1,303 (27)	1,303 (27)	1,295 (25)	326 (8)	8 (4)	0 (0)
25	pfSense	256 (60)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
26	Polycom	644 (6)	24 (1)	7 (1)	7 (1)	0 (0)	0 (0)	0 (0)
27	QNAP	464 (88)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
28	RouterTech	12 (N/A)	12 (N/A)	0 (0)	12 (N/A)	0 (0)	0 (0)	0 (0)
29	Seiki	16 (10)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
30	Supermicro	150 (77)	26 (17)	26 (17)	26 (17)	0 (0)	0 (0)	0 (0)
31	Synology	2,094 (170)	181 (51)	34 (12)	16 (12)	0 (0)	0 (0)	0 (0)
32	Tenda	244 (55)	59 (22)	52 (19)	59 (22)	1 (1)	1 (1)	0 (0)
33	Tenvis	49 (4)	26 (3)	26 (3)	26 (3)	17 (3)	17 (3)	0 (0)
34	Thuraya	18 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
35	Tomato by Shibby	2,942 (6)	2,940 (6)	2,940 (6)	2,940 (6)	21 (2)	20 (2)	1 (1)
36	TP-Link	1,072 (367)	302 (103)	302 (103)	300 (102)	245 (81)	206 (73)	3 (1)
37	TRENDnet	822 (162)	272 (46)	269 (45)	270 (46)	132 (26)	94 (17)	15 (1)
38	Ubiquiti	51 (11)	36 (8)	25 (5)	36 (8)	0 (0)	0 (0)	0 (0)
39	u-blox	16 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
40	Verizon	37 (1)	2 (N/A)	1 (N/A)	2 (N/A)	0 (0)	0 (0)	0 (0)
41	Western Digital	1 (N/A)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
42	ZyXEL	1,768 (252)	161 (38)	159 (38)	159 (39)	59 (18)	37 (13)	5 (1)
Total	42	23,035 (2,331)	8,893 (667)	8,617 (611)	8,591 (625)	2,797 (322)	1,971 (252)	887 (89)

TABLE VII: Breakdown of analysis progress by vendor, in terms of firmware images (products). Note: `N/A` indicates that we do not have sufficient metadata to compute a lower-bound on affected products.