A Large-Scale Analysis of the Security of Embedded Firmwares

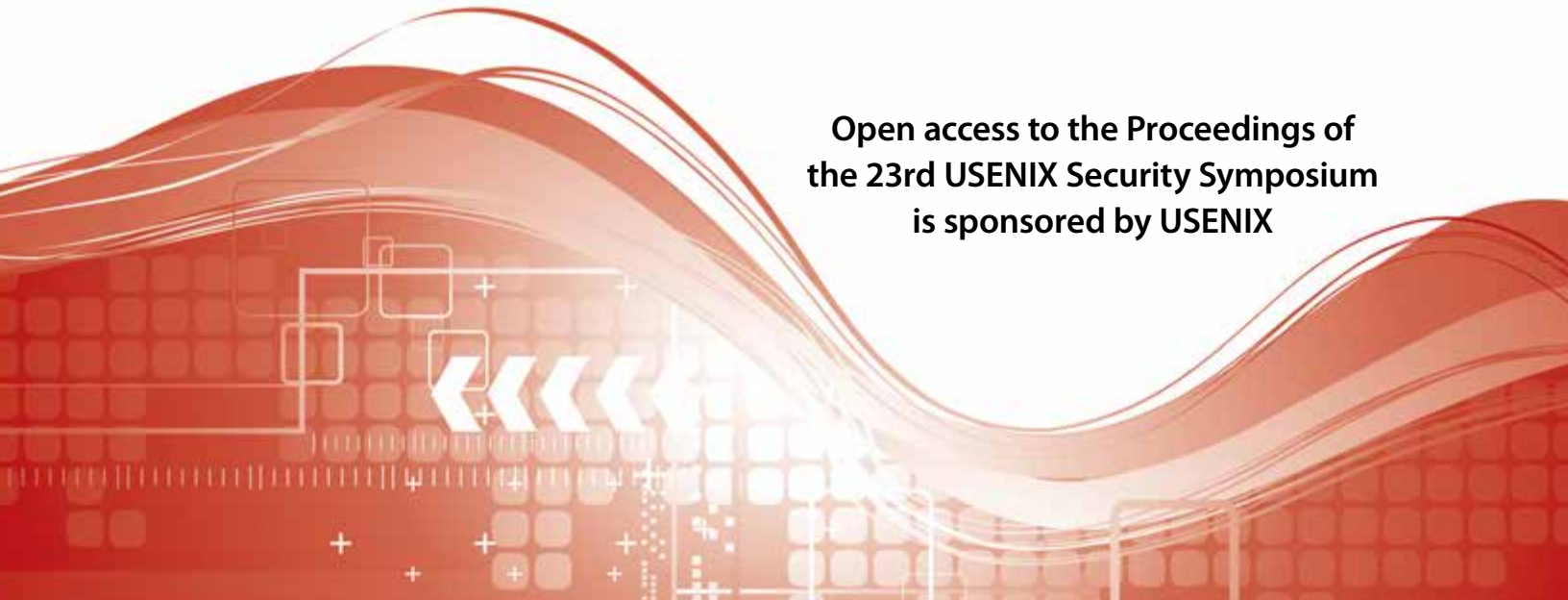
Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti, *Eurecom*

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7



Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX

A Large-Scale Analysis of the Security of Embedded Firmwares

Andrei Costin, Jonas Zaddach, Aurélien Francillon and Davide Balzarotti

EURECOM
Sophia Antipolis
France
{*name.surname*}@*eurecom.fr*

Abstract

As embedded systems are more than ever present in our society, their security is becoming an increasingly important issue. However, based on the results of many recent analyses of individual firmware images, embedded systems acquired a reputation of being insecure. Despite these facts, we still lack a global understanding of embedded systems' security as well as the tools and techniques needed to support such general claims.

In this paper we present the first public, large-scale analysis of firmware images. In particular, we unpacked 32 thousand firmware images into 1.7 million individual files, which we then statically analyzed. We leverage this large-scale analysis to bring new insights on the security of embedded devices and to underline and detail several important challenges that need to be addressed in future research. We also show the main benefits of looking at many different devices at the same time and of linking our results with other large-scale datasets such as the ZMap's HTTPS survey.

In summary, without performing sophisticated static analysis, we discovered a total of 38 previously unknown vulnerabilities in over 693 firmware images. Moreover, by correlating similar files inside apparently unrelated firmware images, we were able to extend some of those vulnerabilities to over 123 different products. We also confirmed that some of these vulnerabilities altogether are affecting at least 140*K* devices accessible over the Internet. It would not have been possible to achieve these results without an analysis at such wide scale.

We believe that this project, which we plan to provide as a firmware unpacking and analysis web service[1], will help shed some light on the security of embedded devices.

[1] `http://firmware.re`

1 Introduction

Embedded systems are omnipresent in our everyday life. For example, they are the core of various Common-Off-The-Shelf (COTS) devices such as printers, mobile phones, home routers, and computer components and peripherals. They are also present in many devices that are less consumer oriented such as video surveillance systems, medical implants, car elements, SCADA and PLC devices, and basically anything we normally call *electronics*. The emerging phenomenon of the Internet-of-Things (IoT) will make them even more widespread and interconnected.

All these systems run special software, often called *firmware*, which is usually distributed by vendors as *firmware images* or *firmware updates*. Several definitions for *firmware* exist in the literature. The term was originally introduced to describe the CPU microcode that existed "somewhere" between the hardware and the software layers. However, the word quickly assumed a broader meaning, and the IEEE Std 610.12-1990 [6] extended the definition to cover the *"combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device"*.

Nowadays, the term *firmware* is more generally used to describe the software that is embedded in a hardware device. Like traditional software, embedded devices' firmware may have bugs or misconfigurations that can result in vulnerabilities for the devices which run that particular code. Due to anecdotal evidence, embedded systems acquired a bad security reputation, generally based on case by case experiences of failures. For instance, a car model throttle control fails [47] or can be maliciously taken over [21, 55]; a home wireless router is found to have a backdoor [48, 7, 44], just to name a few recent examples. On the one hand, apart from a few works that targeted specific devices or software versions [39, 27, 63], to date there is still no large-scale security analysis of firmware images. On the other hand,

manual security analysis of firmware images yields very accurate results, but it is extremely slow and does not scale well for a large and heterogeneous dataset of firmware images. As useful as such individual reports are for a particular device or firmware version, these alone do not allow to establish a general judgment on the overall state of the security of firmware images. Even worse, the same vulnerability may be present in different devices, which are left vulnerable until those flaws are rediscovered independently by other researchers [48]. This is often the case when several *integration vendors* rely on the same subcontractors, tools, or SDKs provided by *development vendors*. Devices may also be branded under different names but may actually run either the same or similar firmware. Such devices will often be affected by exactly the same vulnerabilities, however, without a detailed knowledge of the internal relationships between those vendors, it is often impossible to identify such similarities. As a consequence, some devices will often be left affected by known vulnerabilities even if an updated firmware is available.

1.1 Methodology

Performing a large-scale study of the security of embedded devices by actually running the physical devices (i.e., using a dynamic analysis approach) has several major drawbacks. First of all, physically acquiring thousands of devices to study would be prohibitively expensive. Moreover, some of them may be hard to operate outside the system for which they are designed — e.g., a throttle control outside a car. Another option is to analyze existing online devices as presented by Cui and Stolfo [29]. However, some vulnerabilities are hard to find by just looking at the running device, and it is ethically questionable to perform any nontrivial analysis on an online system without authorization.

Unsurprisingly, static analysis scales better than dynamic analysis as it does not require access to the physical devices. Hence, we decided to follow this approach in our study. Our methodology consists of collecting firmware images for as many devices and vendors as possible. This task is complicated by the fact that firmware images are diverse and it is often difficult to tell firmware images apart from other files. In particular, distribution channels, packaging formats, installation procedures, and availability of meta-data often depend on the vendor and on the device type. We then designed and implemented a distributed architecture to unpack and run simple static analysis tasks on the collected firmware images. However, the contribution of this paper is not in the static analysis techniques we use (for example, we did not perform any static *code* analysis), but to show the advantages of an horizontal, large-scale exploration. For this reason, we implemented a correlation engine to compare and find similarities between all the objects in our dataset. This allowed us to quickly "propagate" vulnerabilities from known vulnerable devices to other systems that were previously not known to be affected by the same vulnerability.

Most of the steps performed by our system are conceptually simple and could be easily performed manually on a few devices. However, we identified *five major challenges* that researchers need to address in order to perform large scale experiments on thousands of different firmware images. These include the problem of building a representative dataset (Challenge A in Section 2), of properly identifying individual firmware images (Challenge B in Section 2), of unpacking custom archive formats (Challenge C in Section 2), of limiting the required computation resources (Challenge D in Section 2), and finally of finding an automated way to confirm the results of the analysis (Challenge E in Section 2). While in this paper we do not propose a complete solution for all these challenges, we discuss the way and the extent to which we dealt with some of these challenges to perform a systematic, automated, large-scale analysis of firmware images.

1.2 Results Overview

For our experiments we collected an initial set of `759,273` files (totaling `1.8TB` of storage space) from publicly accessible firmware update sites. After filtering out the obvious noise, we were left with `172,751` potential firmware images. We then sampled a set of `32,356` firmware candidates that we analyzed using a private cloud deployment of `90` worker nodes. The analysis and reports resulted in a `10GB` database.

The analysis of sampled files led us to automatically discover and report 38 new vulnerabilities (fixes for some of these are still pending) and to confirm several that were already known [44, 48]. Some of our findings include:

- We extracted private RSA keys and their self-signed certificates used in about 35,000 online devices (mainly associated with surveillance cameras).
- We extracted several dozens of hard-coded password hashes. Most of them were weak, and therefore we were able to easily recover the original passwords.
- We identified a number of possible backdoors such as the `authorized_keys` file (which lists the SSH keys that are allowed to remotely connect to the system), a number of hard-coded `telnetd` credentials affecting at least 2K devices, hard-coded web-login admin credentials affecting at least 101K de-

vices, and a number of backdoored daemons and web pages in the web-interface of the devices.
- Whenever a new vulnerability was discovered (by other researchers or by us) our analysis infrastructure allowed us to quickly find related devices or firmware versions that were likely affected by the same vulnerability. For example, our *correlation techniques* allowed us to correctly extend the list of affected devices for variations of a `telnetd` hard-coded credentials vulnerability. In other cases, this led us to find a vulnerability's root problem spread across multiple vendors.

1.3 Contributions

In summary this paper makes the following contributions:

- We show the advantages of performing a large-scale analysis of firmware images and describe the main challenges associated with this activity.
- We propose a framework to perform firmware collection, filtering, unpacking and analysis at large scale.
- We implemented several efficient static techniques that we ran on 32,356 firmware candidates.
- We present a correlation technique which allows to propagate vulnerability information to similar firmware images.
- We discovered 693 firmware images affected by at least one vulnerability and reported 38 new CVEs.

2 Challenges

As mentioned in the previous section, there are clear advantages of performing a wide-scale analysis of embedded firmware images. In fact, as is often the case in system security, certain phenomena can only be observed by looking at the global picture and not by studying a single device (or a single family of devices) at a time.

However, large-scale experiments require automated techniques to obtain firmware images, unpack them, and analyze the extracted files. While these are easy tasks for a human, they become challenging when they need to be fully automated. In this section we summarize the five main challenges that we faced during the design and implementation of our experiments.

Challenge A: Building a Representative Dataset

The embedded systems environment is heterogeneous, spanning a variety of devices, vendors, architectures, instruction sets, operating systems, and custom components. This makes the task of compiling a *representative* and *balanced* dataset of firmware images a difficult problem to solve.

The real market distribution of a certain hardware architecture is often unknown, and it is hard to compare different classes of devices (e.g., medical implants vs. surveillance cameras). Which of them need to be taken into account to build a representative firmware dataset? How easy is it to generalize a technique that has only been tested on a certain brand of routers to other vendors? How easy is it to apply the same technique to other classes of devices such as TVs, cameras, insulin pumps, or power plant controllers?

From a practical point of view, the lack of centralized points of collection (such as the ones provided by antivirus vendors or public sandboxes in the malware analysis field) makes it difficult for researchers to gather a large and well triaged dataset. Firmware often needs to be downloaded from the vendor web pages, and it is not always simple, even for a human, to tell whether or not two firmware images are for the same physical device.

Challenge B: Firmware Identification

One challenge often encountered in firmware analysis and reverse engineering is the difficulty of reliably extracting meta-data from a firmware image. For instance, such meta-data includes the vendor, the device product code and purpose, the firmware version, and the processor architecture, among many other details.

In practice, the diversity of firmware file formats makes it harder to even recognize that a given file downloaded from a vendor website is a firmware at all. Often firmware updates come in unexpected formats such as *HP Printer Job Language* and *PostScript* documents for printers [24, 23, 27], *DOS executables* for BIOS, and *ISO images* for hard disk drives [72].

In many cases, the only source of reliable information is the official vendor documentation. While this is not a problem when looking manually at a few devices, extending the analysis to hundreds of vendors and thousands of firmware images automatically downloaded from the Internet is challenging. In fact, the information retrieval process is hard to automate and is error prone, in particular for certain classes of meta-data. For instance, we often found it hard to infer the correct version number. This makes it difficult for a large-scale collection and analysis system to tell which is the latest version available for a certain device, and even if two firmware images corresponded to different versions for the same device. This further complicates the task of building an unbiased dataset.

Challenge C: Unpacking and Custom Formats

Assuming the analyst succeeded in collecting a representative and well labeled dataset of firmware images, the next challenge consists in locating and extracting important functional blocks (e.g., binary code, configuration files, scripts, web interfaces) on which static analysis routines can be performed.

While this task would be easy to address for traditional software components, where standardized formats for the distribution of machine code (e.g., PE and ELF), resources (e.g., JPEG and GZIP) and groups of files (e.g., ZIP and TAR) exist, embedded software distribution lacks standards. Vendors have developed their own file formats to describe flash and memory images. In some cases those formats are compressed with non-standard compression algorithms. In other cases those formats are obfuscated or encrypted to prevent analysis. Monolithic firmware, in which the bootloader, the operating system kernel, the applications, and other resources are combined together in a single memory image are especially challenging to unpack.

Forensic strategies, like file *carving*, can help to extract known file formats from a binary blob. Unfortunately those methods have drawbacks: On the one hand, they are often too aggressive with the result of extracting data that matches a file pattern only by chance. On the other hand, they are computationally expensive, since each unpacker has to be tried for each file offset of the binary firmware blob.

Finally, if a binary file has been extracted that does not match any known file pattern, it is impossible to say if this file is a data file, or just another container format that is not recognized by the unpacker. In general, we tried to unpack at least until reaching uncompressed files. In some cases, our extraction goes one step further and tries to extract sections, resources and compressed streams (e.g., for the ELF file format).

Challenge D: Scalability and Computational Limits

One of the main advantages of performing a wide-scale analysis is the ability of correlating information across multiple devices. For example, this allowed us to automatically identify the re-use of vulnerable components among different firmware images, even from different vendors.

Capturing the global picture of the relationship between firmware images would require the one-to-one comparison of each pair of unpacked files. Fuzzy hashes (such as sdhash [62] and ssdeep [54]) are a common and effective solution for this type of task and they have been successfully used in similar domains, e.g., to correlate samples that belong to the same malware families [35, 15]. However, as described in more detail in Section 3.4, computing the similarity between the objects extracted from 26,275 firmware images requires 10^{12} comparisons. Using the simpler fuzzy hash variant, we estimate that on a single dual-core computer this task would take approximately 850 days[2]. This simple estimation highlights one of the possible *computational challenges* associated with a large-scale firmware analysis. Even if we had a perfect database design and a highly optimized in-memory database, it would still be hard to compute, store, and query the fuzzy hash scores of all pairs of unpacked files. A distributed computational infrastructure can help reduce the total time since the task itself is parallelizable [57]. However, since the number of comparisons grows quadratically with the number of elements to compare, this problem quickly becomes impracticable for large image datasets. If, for example, one would like to build a fuzzy hash database for our whole dataset, which is just five times the size of the current sampled dataset, this effort would already take more than 150 CPU years instead of 850 CPU days. Our attempt to use the GPU-assisted fuzzy hashing provided by `sdhash` [62] only resulted in a limited speedup that was not sufficient to perform a full-scale comparison of all files in our dataset.

Challenge E: Results Confirmation

The first four challenges were mostly related to the collection of the dataset and the pre-processing of the firmware images. Once the code or the resources used by the embedded device have been successfully extracted and identified, researchers can focus their attention on the static analysis. Even though the details and goals of this step are beyond the scope of this paper, in Section 3.3 we present some examples of simple static analysis and we discuss the advantages of performing these techniques on a large scale.

However, one important research challenge remains regarding the way the results of static analysis can be confirmed. For example, we can consider a scenario where a researcher applies a new vulnerability detection technique to several thousand firmware images. Those images were designed to run on specific embedded devices, most of which are not available to the researcher and would be hard and costly to acquire. Lacking the proper hardware platform, there is still no way to manually or automatically test the *affected code* to confirm or deny the findings of the static analysis.

For example, in our experiments we identified a firmware image that included the `PHP 5.2.12` banner string. This allowed us to easily identify several vulnerabilities

[2] This is mainly because comparing fuzzy hashes is not a simple bit string comparison but actually involves a rather complex algorithm and high computational effort.

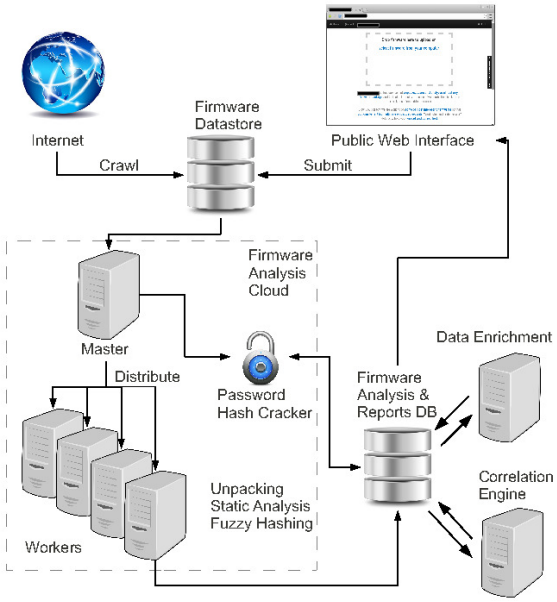


Figure 1: Architecture of the entire system.

associated with that version of the PHP interpreter. However, this is insufficient to determine if the PHP interpreter is vulnerable, since the vendor may have applied patches to correct known vulnerabilities without this being reflected in the version string. In addition, the vendor might have used an architecture and/or a set of compilation options which produced a non-vulnerable build of the component. Unfortunately, even if a proof of concept attack exists for that vulnerability, without the proper hardware it is impossible to test the firmware and confirm or deny the presence of the problem.

Confirming the results of the static analysis on firmware devices is a tedious task requiring manual intervention from an expert. Scaling this effort to thousands of firmware images is even harder. Therefore, we believe the development of new techniques is required to accurately deal with this problem at a large scale.

3 Setup

In this section we first present the design of our distributed static analysis and correlation system. Then we detail the techniques we used, and how we addressed the challenges described in Section 2.

3.1 Architecture

Figure 1 presents an overview of our architecture. The first component of our analysis platform is the *firmware data store*, which stores the unmodified firmware files that have been retrieved either by the *web crawler* or that have been submitted through the public *web interface*. When a new file is received by the firmware data store, it is automatically scheduled to be processed by the *analysis cloud*. The analysis cloud consists of a master node, and a number of worker and hash cracking nodes. The *master node* distributes unpacking jobs to the *worker nodes* (Figure 2), which unpack and analyze firmware images. *Hash cracking nodes* process password hashes that have been found during the analysis, and try to find the corresponding plaintext passwords. Apart from coordinating the worker nodes, the master node also runs the *correlation engine* and the *data enrichment system* modules. These modules improve the reports with results from the cross-firmware analysis.

The analysis cloud is where the actual analysis of the firmware takes place. Each firmware image is first submitted to the *master node*. Subsequently, *worker nodes* are responsible for unpacking and analyzing the firmware and for returning the results of the analysis back to the master node. At this point, the master node will submit this information to the *reports database*. If there were any uncracked password hashes in the analyzed firmware, it will additionally submit those hashes to one of the *hash cracking nodes* which will try to recover the plaintext passwords.

It is important to note that only the results of the analysis and the meta-data of the unpacked files are stored in the database. Even though we do not currently use the extracted files after the analysis, we still archive them for future work, or in case we want to review or enhance a specific set of analyzed firmware images.

The architecture contains two other components: the *correlation engine* and the *data enrichment system*. Both of them fetch the results of the firmware analysis from the reports database and perform additional tasks. The correlation engine identifies a number of “interesting” files and tries to correlate them with any other file present in the database. The enrichment system is responsible for enhancing the information about each firmware image by performing online scans and lookup queries (e.g., detecting vendor name, device name/code and device category).

In the remainder of this section we describe each step of the firmware analysis in more detail so that our experiments can be reproduced.

3.2 Firmware Acquisition and Storage

The first step of our experiments consisted in gathering a firmware collection for analysis. We achieved this goal by using mainly two methods: a web crawler that automatically downloads files from manufacturers’ websites and specialized mirror sites, and a website with a submis-

sion interface where users can submit firmware images for analysis.

We initialized the crawler with tens of support pages from well known manufacturers such as Xerox, Bosch, Philips, D-Link, Samsung, LG, Belkin, etc. Second, we used public FTP indexing engines [3] to search for files with keywords related to firmware images (e.g., `firmware`). The result of such searches yields either directory URLs, which are added to the crawler list of URLs to index and download, or file URLs, which are directly downloaded by the crawler. At the same time, the script strips filenames out of the URLs to create additional directory URLs.

Finally, we used Google Custom Search Engines (GCSE) [3] to create customized search engines. GCSE provides a flexible API to perform advanced search queries and returns results in a structured way. It also allows to programmatically create a very customized CSE on-the-fly using a combination of RESTful and XML APIs. For example, a CSE is created using `support.nikonusa.com` as the "Sites to Search" parameter. Then a firmware related query is used on the CSE such as `''firmware download''`. The CSE from the above example returns 2,210 results at the time of this publication. The result URLs along with associated meta-data are retrieved via the JSON API. Each URL was then used by the crawler or as part of other dynamic CSE, as previously described. This allowed us to mine additional firmware images and firmware repositories.

We chose not to filter data at collection time, but to download files greedily, deciding at a later stage if the collected files were firmware images or not. The reason for this decision is two-fold. First, accompanying files such as manuals and user guides can be useful for finding additional download locations or for extracting contained information (e.g., model, default passwords, update URLs). Second, as we mentioned previously, it is often difficult to distinguish firmware images from other files. For this reason, filtering a large dataset is better than taking a chance to miss firmware files during the downloading phase. In total, we crawled 284 sites and stopped downloading once the collection of files reached 1.8TB of storage. The actual storage required for this amount of data is at least 3-4 times larger, since we used mirrored backup storage, as well as space for keeping the unpacked files and files generated during the unpacking (e.g., logs and analysis results).

The public *web submission interface* provides a means for security researchers to submit firmware files for analysis. After the analysis is completed, the platform produces a report with information about the firmware contents as well as similarities to other firmware in our database. We have already received tens of firmware images through the submission interface. While this is currently a marginal source of firmware files, we expect that more firmware will be submitted as we advertise our service. This will also be a unique chance to have access to firmware images that are not generally available and, for example, need to be manually extracted from a device.

Files fetched by the web crawler and received from the web submission interface are added to the *firmware data store*. Files are simply stored on a file system and a database is used for meta-data (e.g., file checksum, size, download location).

[3] FTP indexing engines such as: `www.mmnt.ru`, `www.filemare.com`, `www.filewatcher.com`, `www.filesearching.com`, `www.ftpsearch.net`, `www.search-ftps.com`

3.3 Unpacking and Analysis

The next step towards the analysis of a firmware image is to unpack and extract the contained files or objects. The output of this phase largely depends on the type of firmware. In some examples, executable code and resources (such as graphics files or HTML code) can be linked into a binary blob that is designed to be directly copied into memory by a bootloader and then executed. Some other firmware images are distributed in a compressed and obfuscated file which contains a block-by-block copy of a flash image. Such an image may consist of several partitions containing a bootloader, a kernel and a file system.

Unpacking Frameworks

There are three main tools to unpack arbitrary firmware images: binwalk [41], FRAK [26] and Binary Analysis Toolkit (BAT) [66].

Binwalk is a well known firmware unpacking tool developed by Craig Heffner [41]. It uses pattern matching to locate and carve files from a binary blob. Additionally, it also extracts meta-data such as license strings.

FRAK is an unpacking toolkit first presented by Cui et al. [27]. Even though the authors mention that the tool would be made publicly available, we were not able to obtain a copy. We therefore had to evaluate its unpacking performance based on the device vendors and models that FRAK supports, according to [27]. We estimated that FRAK would have unpacked less than 1% of the files we analyzed, while our platform was able to unpack more than 81% of them. This said, both would be complementary as some of the file formats FRAK unpacks are unsupported by our tool at present.

The Binary Analysis Toolkit (BAT), formerly known as GPLtool, was originally designed by Tjaldur software to detect GPL violations [45, 66]. To this end, it recursively extracts files from a firmware blob and matches strings with a database of known strings from

Table 1: Comparison of Binwalk, BAT, FRAK and our framework. The last three columns show if the respective unpacker was able to extract the firmware. Note that this is a non statistically significant sample which is given for illustrating unpacking performance (manual analysis of each firmware is time consuming). As FRAK was not available for testing, its unpacking performance was estimated based on information from [26]. The additional performance of our framework stems from the many customizations we have incrementally developed over BAT (Figure 2).

Device	Vendor	OS	Binwalk	BAT	FRAK	Our framework
PC	Intel	BIOS	✗	✗	✗	✗
Camera	STL	Linux	✗	✓	✗	✓
Router	Bintec	-	✗	✗	✗	✗
ADSL Gateway	Zyxel	ZynOS	✓	✓	✗	✓
PLC	Siemens	-	✓	✓	✗	✓
DSLAM	-	-	✓	✓	✗	✓
PC	Intel	BIOS	✓	✓	✗	✓
ISDN Server	Planet	-	✓	✓	✗	✓
Voip	Asotel	Vxworks	✓	✓	✗	✓
Modem	-	-	✗	✗	✗	✓
Home Automation	Belkin	Linux	✗	✗	✗	✓
			55%	64%	0%	82%

GPL projects. Additionally, BAT supports file carving similar to binwalk.

Table 1 shows a simple comparison of the unpacking performance of each framework on a few samples of firmware images. We chose to use BAT because it is the most complete tool available for our purposes. It also has a significantly lower rate of false positive extractions compared to binwalk. In addition, binwalk did not support recursive unpacking at the time when we decided on an unpacking framework. Nevertheless, the interface between our framework and BAT has been designed to be generic so that integrating other unpacking toolkits (such as binwalk) is easy.

We developed a range of additional plugins for BAT. These include plugins which extract interesting strings (e.g., software versions or password hashes), add unpacking methods, gather statistics and collect interesting files such as private key files or `authorized_keys` files. In total we added 35 plugins to the existing framework.

Password Hash Cracking

Password hashes found during the analysis phase are passed to a hash cracking node. These nodes are dedicated physical hosts with a Nvidia Tesla GPU [56] that run a CUDA-enabled [59] version of *John The Ripper* [60]. John The Ripper is capable of brute forcing most encoded password hashes and detecting the type of hash and salt used. In addition to this, a dictionary can be provided to seed the password cracking. For each brute force attempt, we provide a dictionary built from common password lists and strings extracted from firmwares, manuals, *readme* files and other resources. This allows to find both passwords that are directly present in those files as well as passwords that are weak and based on keywords related to the product.

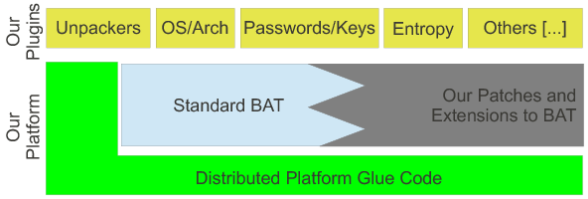


Figure 2: Architecture of a single worker node.

Parallelizing the Unpacking and Analysis

To accelerate the unpacking process, we distributed this task on several worker nodes. Our distributed environment is based on the *distributed-python-for-scripting* framework [65]. Data is synchronized between the repository and the nodes using *rsync (over ssh)* [67].

Our loosely coupled architecture allows us to run worker nodes virtually anywhere. For instance, we instantiated worker virtual machines on a local VMware server and several OpenStack servers, as well as on Amazon EC2 instances. At the time of this publication we were using 90 such virtual machines to analyze firmware files.

3.4 Correlation Engine

The unpacked firmware images and analysis results are stored into the *analysis & reports database*. This allows us to perform queries, to generate reports and statistics, and to easily integrate our results with other external components. The correlation engine is designed to find similarities between different firmware images. In particular, the comparison is made along four different dimensions: shared credentials, shared self-signed certificates, common keywords, and fuzzy hashes of the firmwares and objects within the firmwares.

Shared Credentials and Self-Signed Certificates

Shared credentials (such as hard coded *non-trivial* passwords) and shared self-signed certificates are effective in finding strong connections between different firmware images of the same vendor, or even firmwares of different vendors. For example, we were able to correlate two brands of CCTV systems based on a common *non-trivial* default password.

Therefore, finding a password of one vendor's product can directly impact the security of others. We also found a similar type of correlation for two other CCTV vendors that we linked through the same self-signed certificate, as explained in Section 5.2.

Keywords

Keywords correlation is based on specific strings extracted by our static analysis plugins. In some cases, for example in Section 5.1, the keyword "`backdoor`" revealed several other keywords. By using the extended set of keywords we clustered several vendors prone to the same backdoor functionality, possibly affecting $500,000$ devices. In other cases, files inside firmware images contain compilation and SDK paths. This turns out to be sufficient to cluster firmware images of different devices.

Fuzzy hashes

Fuzzy hash triage (comparison, correlation and clustering) is the most generic correlation technique used by our framework. The engine computes both the *ssdeep* and the *sdhash* of every single object extracted from the firmware image during the unpacking phase. This is a powerful technique that allows us to find files that are "similar" but for which a traditional hash (such as *MD5* or *SHA1*) would not match. Unfortunately, as we already mentioned in Section 2, a complete one-to-one comparison of fuzzy hashes is currently infeasible on a large scale. Therefore, we compute the fuzzy hashes of each file that was successfully extracted from a firmware image and store this result. When a file is found to be interesting we perform the fuzzy hash comparison between this file's hash and all stored hashes.

For example, a file (or all files unpacked from a firmware) may be flagged as interesting because it is affected by a known vulnerability, or because we found it to be vulnerable by static analysis. If another firmware contains a file that is similar to a file from a vulnerable firmware, then there might be a chance that the first firmware is also vulnerable. We present such an example in Section 5.3, where this approach was successful and allowed us to propagate known vulnerabilities of one device to other similar devices of *different* vendors.

Future work

In the literature, there are several approaches proposed to perform comparison, clustering, and triage on a large scale. Jang et al. propose large-scale triage techniques of PC malware in BitShred [52]. The authors concluded that at the rate of 8,000 unique malware samples per day, which required 31M comparisons, it is unfeasible on a single CPU to perform one-to-one comparisons to find malware families using hierarchical clustering. French and Casey [13] propose, before fuzzy hash comparison, to perform a "bins" partitioning approach based on the block and file sizes. This approach, for their particular dataset and bins partitioning strategy, allowed on average to reduce the search space for a given fuzzy hash down to 16.9%. Chakradeo et al. [20] propose MAST, an effective and well performing triage architecture for mobile market applications. It solves the manual and resource-intensive automated analysis at market-scale using Multiple Correspondence Analysis (MCA) statistical method.

As a future work, there are several possible improvements to our approach. For instance, instead of performing all comparisons on a single machine, we could adopt a distributed comparison and clustering infrastructure, such as the Hadoop implementation of MapReduce [32] used by BitShred. Second, on each comparison and clustering node we could use the "bins" partitioning approach from French and Casey [13].

3.5 Data Enrichment

The data enrichment phase is responsible for extending the knowledge base about firmware images, for example by performing automated queries and passive scans over the Internet. In the current prototype, the data enrichment relies on two simple techniques. First, it uses the `<title>` tag of web pages and `authentication realms` of web servers when these are detected inside a firmware. This information is then used to build targeted *search queries* (such as *"intitle:Router ABC-123 Admin Page"*) for both Shodan [5] and GCSE.

Second, we correlate SSL certificates extracted from firmware images to those collected by the ZMap project. ZMap was used in [37] to scan the whole IPv4 address space on the 443 port, collecting SSL certificates in a large database.

Correlating these two large-scale databases (i.e., ZMap's HTTPS survey and our firmware database) provides new insights. For example, we are able to quickly evaluate the severity of a particular vulnerability by identifying publicly reachable devices that are running a given firmware image. This gives a good estimate for the number of publicly accessible vulnerable devices.

For instance, our framework found 41 certificates having unprotected private keys. Those keys were extracted from firmware images in the unpacking and analysis phase. The data enrichment engine subsequently found the same self-signed certificate in over 35K devices reachable on the Internet. We detail this case study in Section 5.2.

3.6 Setup Development Effort

Our framework relies on many existing tools. In addition to this, we have put a considerable effort (over 20k lines of code according to `sloccount` [68]) to extend BAT, develop new unpackers, create the results analysis platform and run results interpretation.

4 Dataset and Results

In this section we describe our dataset and we present the results of the global analysis, including the discussion of the new vulnerabilities and the common bad practices we discovered in our experiments. In Section 5, we will then present a few concrete case studies, illustrating how such a large dataset can provide new insights into the security of embedded systems.

4.1 General Dataset Statistics

While we currently collect firmware images from multiple sources, most of the images in our dataset have been downloaded by crawling the Internet. As a consequence, our dataset is biased towards devices for which firmware updates can be found online, and towards known vendors that maintain well organized websites.

We also decided to exclude firmware images of smartphones from our study. In fact, popular smartphone firmware images are complete operating system distributions, most of them iOS, Android or Windows based – making them closer to general purpose systems than to embedded devices.

Our crawler collected 759,273 files, for a total of 1.8TB of data. After filtering out the files that were clearly unrelated (e.g., manuals, user guides, web pages, empty files) we obtained a dataset of 172,751 files. Our architecture is constantly running to fetch more samples and analyze them in a distributed fashion. At the time of this publication the system was able to process (unpack and analyze) 32,356 firmware images.

Firmware Identification The problem of properly identifying a firmware image (Challenge 2) still requires a considerable amount of manual effort. Doing so accurately and automatically at a large scale is a daunting task. Nevertheless, we are interested in having an estimate of the number of actual firmware images in our dataset.

For this purpose we manually analyzed a number of random samples from our dataset of 172,751 potential firmware images and computed a *confidence interval* [19] to estimate the global representativeness in the dataset. In particular, after manually analyzing 130 random files from the total of 172,751, we were able to mark only 44 as firmware images. This translates to a proportion of 34% ($\pm$ 8%) firmware images on our dataset – with a 95% confidence. The manual analysis process took approximately one person-week because the inspection of the extracted files for firmware code is quite tedious.

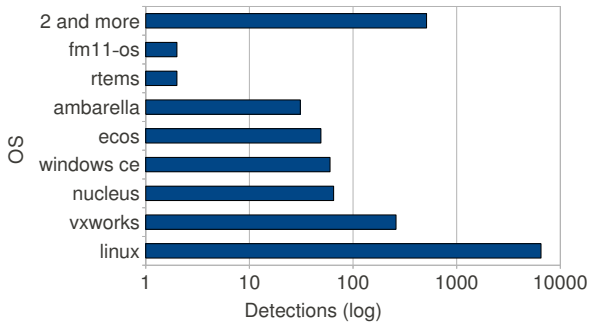


Figure 3: OS distribution among firmware images.

We can therefore expect our dataset to contain between 44,431 and 72,520 firmware images (by applying 34%−8%, and 34%+8% respectively, to the entire candidates set of 172,751). While the range is still relatively large, this estimation gives a 95% reliable measure of the useful data in our sample. We also developed a heuristic to automatically detect if a file is successfully unpacked or not. This heuristic takes multiple parameters, such as the number, type and size of files carved out from a firmware, into account. Such an empirical heuristic is not perfect, but it can guide our framework to mark a file as unpacked or not, and then take actions accordingly.

Files Analysis As described in Section 3.3, unpacking unknown files is an error-prone and time-consuming task. In fact, when the file format is not recognized, unpacking relies on a slow and imprecise carving approach. File carving is essentially an attempt to unpack at every offset of the file, iterating over several known signatures (e.g., archive magic headers).

As a result, out of the 32,356 files we processed so far, 26,275 were successfully unpacked. The process is nevertheless continuous and more firmware images are being unpacked over time.

4.2 Results Overview

In the rest of the section we present the results of the analysis performed by our plugins right after each firmware image was unpacked.

Files Formats The majority of initial files being unpacked were identified as *compressed files* or *raw data*. Once unpacked, most of those firmware images were

identified as targeting ARM (63%) devices, followed by MIPS (7%). As reported in Figure 3, Linux is the most frequently encountered embedded operating system in our dataset – being present in more than three quarters (86%) of all analyzed firmware images. The remaining images contain proprietary operating systems like VxWorks, Nucleus RTOS and Windows CE, which altogether represent around 7%. Among Linux based firmware images, we identified 112 distinct Linux kernel versions.

Password Hashes Statistics Files like `/etc/passwd` and `/etc/shadow` store hashed versions of account credentials. These are usual targets for attackers since they can be used to retrieve passwords which often allow to login remotely to a device at a later time. Hence, an analysis of these files can help understanding how well an embedded device is protected.

Our plugin responsible for collecting entries from `/etc/passwd` and `/etc/shadow` files retrieved 100 distinct password hashes, covering 681 distinct firmware images and belonging to 27 vendors. We were also able to recover the plaintext passwords for 58 of those hashes, which occur in 538 distinct firmware images. The most popular passwords were `<empty>`, `pass`, `logout`, and `helpme`. While these may look trivial, it is important to stress that they are actually used in a large number of embedded devices.

Certificates and Private RSA Keys Statistics Many vendors include self-signed certificates inside their firmware images [43, 42]. Due to bad practices in both *release management* and *software design*, some vendors also include the private keys (e.g., PEM, GPG), as confirmed by recent advisories [49, 51].

We developed two simple plugins for our system which collect SSL certificates and private keys. These plugins also collect their fingerprints and check for empty or trivial passphrases. So far, we have been able to extract 109 private RSA keys from 428 firmware images and 56 self-signed SSL certificates out of 344 firmware images. In total, we obtained 41 self-signed SSL certificates together with their corresponding private RSA keys. By looking up those certificates in the public ZMap datasets [36], we were able to automatically locate about 35,000 active online devices.

For all these devices, if the certificate and private key are not regenerated on the first boot after a firmware update, HTTPS encryption can be easily decrypted by an attacker by simply downloading a copy of the firmware image. In addition, if both a regenerated and a firmware-shipped self-signed certificate are used interchangeably, the user of the device may still be vulnerable to man-in-the-middle (MITM) attacks.

Packaging Outdated and Vulnerable Software Another interesting finding relates to bad *release management* by embedded firmware vendors. Firmware images often rely on many third-party software and libraries. Those keep updating and have security fixes every now and then. OWASP Top Ten [61] lists *"Using Components with Known Vulnerabilities"* at position nine and underlines that *"upgrading to these new versions is critical"*.

In one particular case, we identified a relatively recently released firmware image that contained a kernel (version 2.4.20) that was built and packaged ten years after its initial release. In another case, we discovered that some recently released firmware images contained nine years old BusyBox versions.

Building Images as *root* While prototyping, putting together a build environment as fast as possible is very important. Unfortunately, sometimes the easiest solution is just to setup and run the entire toolchains as superuser.

Our analysis plugins extracted several compilation banners such as `Linux version 2.6.31.8-mv78100 (root@ubuntu) (gcc version 4.2.0 20070413 (prerelease)) Mon Nov 7 16:51:58 JST 2011` or `BusyBox v1.7.0 (2007-10-15 19:49:46 IST)`.

24% of the 450 unique banners we collected containing the `user@host` combinations were associated to the `root` user. In addition to this, among the 267 unique hostnames extracted from those banners, *ten* resolved to public IP addresses and *one* of these even accepted incoming SSH connections.

All these findings reveal a number of unsafe practices ranging from *build management* (e.g., build process done as `root`) to *infrastructure management* (e.g., build hosts reachable over public networks), to *release management* (e.g., usernames and hostnames not removed from production release builds).

Web Servers Configuration We developed plugins to analyze the configuration files of web servers embedded in the firmware images such as `lighttpd.conf` or `boa.conf`. We then parsed the extracted files to retrieve specific configuration settings such as the running user, the documents root directory, and the file containing authentication secrets. We collected in total 847 distinct web server configuration files and the findings were discouraging. We found that in more than 81% of the cases the web servers were configured to run as a privileged user (i.e., having a setting such as `user=root`). This reveals unsafe practices of insecure design and configuration. Running the web server of an embedded device with unnecessarily high privileges can be extremely risky since the security of the entire device can be compromised by finding a vulnerability in one of the web components.

5 Case Studies

5.1 Backdoors in Plain Sight

Many backdoors in embedded systems have been reported recently, ranging from very simple cases [44] to others that were more difficult to discover [50, 64]. In one famous case [44], the backdoor was found to be activated by the string "`xmlset_roodkcableoj28840ybtide`" (i.e., `edit by 04882 joel backdoor` in reverse). This fully functional backdoor was affecting three vendors. Interestingly enough, this backdoor may have been detected earlier by a simple keyword matching on the open source release from the vendor[2].

Inspired by this case, we performed a string search in our dataset with various backdoor related keywords. Surprisingly, we found 1198 matches, in 326 firmware candidates.

Among those search results, several matched the firmware of a home automation device from a major vendor. According to download statistics from Google Play and Apple App Store, more than half a million users have downloaded an app for this device [9, 8].

We manually analyzed the firmware of this Linux-based embedded system and found that a daemon process listens on a network multicast address. This service allows execution of remote commands with `root` privileges without any authentication to anybody in the local network. An attacker can easily gain full control if he can send multicast packets to the device.

We then used this example as a *seed* for our *correlation engine*. With this approach we found exactly the same backdoor in two other classes of devices from two different vendors. One of them was affecting 109 firmware images of 44 camera models of a major CCTV solutions vendor, *Vendor C*. The other case is affecting three firmware images for home routers of a major networking equipment vendor, *Vendor D*.

We investigated the issue and found that the affected devices were relying on the same provider of a System on a Chip (SoC) for networking devices. It seems that this backdoor is intended for system debugging, and is part of a development kit. Unfortunately we were not able to locate the source of this binary. We plan to acquire some of those devices to verify the exploitability of the backdoor.

5.2 Private SSL Keys

In addition to the backdoors left in firmware images from *Vendor C*, we also found many firmware images containing public and *private RSA key* pairs. Those unprotected keys are used to provide SSL access to the CCTV camera's web interface. Surprisingly, this private key is the same across many firmware images of the same brand.

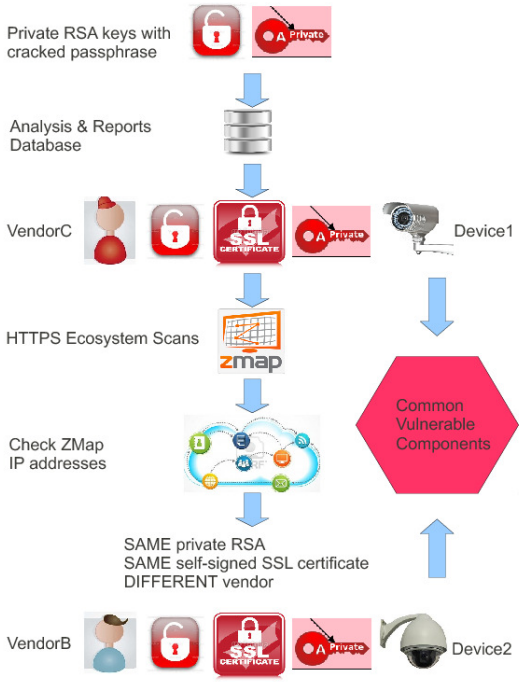


Figure 4: Correlation engine and shared self-signed certificates clustering.

Our platform automatically extracts the fingerprint of the public keys, private keys and SSL certificates. Those keys are then searched in ZMap's HTTPS survey database [36, 37]. *Vendor C*'s SSL certificate was found to be used by around 30K online IP addresses, most likely each corresponding to a single online device. We then fetched the web pages available at those addresses (without trying to authenticate). Surprisingly, we found CCTV cameras branded by another vendor – *Vendor B* – which appears to be an *integrator*. Upon inspection, cameras of *Vendor B* served *exactly the same* SSL certificate as cameras from *Vendor C* (including the SSL *Common Name*, and SSL *Organizational Unit* as well as many other fields of the SSL certificate). The only difference is that CCTV cameras of *Vendor B* returned branded authentication realms, error messages and logos. The *correlation engine* findings are summarized in Figure 4.

Unfortunately, the firmware images from *Vendor B* do not seem to be publicly available. We are planning to obtain a device to extract its firmware and to confirm our findings. We have reported these issues to the vendor. Nevertheless, it is very likely that devices from *Vendor B* are also vulnerable to the multicast packet backdoor given the clear relationship with *Vendor C* that that our platform discovered.

5.3 XSS in WiFi Enabled SD Cards?

SD cards are often more complex than one would imagine. Most SD cards actually contain a processor which runs firmware. This processor often manages functions such as the flash memory translation layer and wear leveling. Security issues have been previously shown on such SD cards [69].

Some SD cards have an embedded WiFi interface with a full fledged web server. This interface allows direct access to the files on the SD card without ejecting it from the device in which it is inserted. It also allows administration of the SD card configuration (e.g., WiFi access points).

We manually found a Cross Site Scripting (XSS) vulnerability in one of these web interfaces, which consists of a `perl` based web application. As this web application does not have platform specific binary bindings, we were able to load the files inside a similar Boa web server on a PC and confirm the vulnerability.

Once we found the exact `perl` files responsible for the XSS, we used our correlation engine based on fuzzy hashes. With this we automatically found another SD card firmware that is vulnerable to the same XSS. Even though the `perl` files were slightly different, they were clearly identified as similar by the fuzzy hash. This correlation would not have been detected by a normal checksum or by a regular hash function.

The process is visualized in Figure 5. The file (*) was found vulnerable. Subsequently, we identified correlated files based on fuzzy hashing. Some of them were related to the same firmware or a previous version of the firmware of the *same* vendor (in red). Also, fuzzy hash correlation identified a similar file in a firmware from a *different* vendor (in orange) that is vulnerable to the same weakness. It further identified some non-vulnerable or non-related files from other vendors (in green).

Those findings are reported as `CVE-2013-5637` and `CVE-2013-5638`. We were also able to confirm this vulnerability and extend the list of affected versions for one of these vendors.

Such manual vulnerability confirmation does not scale. Hence, in the future we plan to integrate static analysis tools for web applications [30, 11, 53, 38, 1] in our process.

6 Ethical Discussion

Large-scale scans to test for the presence of vulnerabilities often raise serious ethical concerns. Even simple Internet-wide network scans may trigger alerts from intrusion detection systems (IDS) and may be perceived as an attack by the scanned networks.

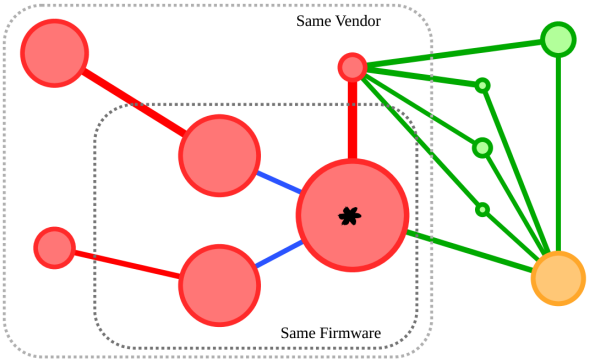


Figure 5: Fuzzy hash clustering and vulnerability propagation. A vulnerability was propagated from a *seed file (*)* to other two files from the same firmware and three files from the same vendor (in red) as well as one file from another vendor (in orange). Also four non-vulnerable files (in green) have a strong correlation with vulnerable files. Edge thickness displays the strength of correlation between files.

In our study we were particularly careful to work within legal and ethical boundaries. First, we obtain firmware images either through user submission or through legitimate distribution mechanisms. In this case, our web crawler was designed to obey the `robots.txt` directives. Second, when we found new vulnerabilities we worked together with vendors and CERTs to confirm the devices vulnerabilities and to perform responsible disclosure. Finally, the license of some firmware images may not allow redistribution. Therefore, the public web submission interface limits the ability to access firmware contents only to the users who uploaded the corresponding firmware image. Other users can only access anonymized reports. We are currently investigating ways to make the full dataset available for research purposes to well identified research institutions.

7 Related Work

Several studies have been proposed to asses the security of embedded devices by scanning the Internet. For instance, Cui et al. [28, 29] present a wide-scale Internet scan to first recognize devices that are known to be shipped with default password, and then to confirm that these devices are indeed still vulnerable by attempting to login into them. Heninger et al. [46] performed the largest ever network survey of TLS and SSH servers, showing that vulnerable keys are surprisingly widespread and that the vast majority appear to belong to headless or embedded devices. ZMap [37] is an efficient and fast network scanner, that allows to scan the complete Internet IPv4 address space in less than one hour. While the scans are not especially targeted to embedded devices, in our work we reuse the SSL certificates

scans performed by ZMap [36]. Similar scans were targeting specific vulnerabilities often present in embedded devices [40, 4]. Such wide-scale scans are mainly targeted at discovering online devices affected by already known vulnerabilities, but in some cases they can help to discover new flaws. However, many categories of flaws cannot be discovered by such scans. Some online services like Shodan [5] provide a global updated view on publicly available devices and web services. This easy-to-use research tool allows security researchers to identify systems worldwide that are potentially exposed or exploitable.

Unpacking firmware images is a known problem, and several tools for this purpose exist. Binwalk [41] is a firmware analysis toolbox that provides various methods and tools for extraction, inspection and reverse engineering of firmware images or other binary blobs. FRAK [26] is a framework to unpack, analyze, and repack firmware images of embedded devices. `FRAK` was never publicly released and reportedly supports only a few firmware formats (e.g., Cisco IP phones and IOS, HP laser printers). The Binary Analysis Toolkit (BAT) [45, 66] was originally designed to detect GPL license violations, mainly by comparing strings in a firmware image to strings present in open source software distributions. For this purpose `BAT` has to unpack firmware images. Unfortunately, as we show in Section 3, none of these tools are accurate and complete enough to be used *as is* in our framework.

There are many examples of security analysis of embedded systems [71]. Several network card firmware images have been analyzed and modified to insert a backdoor [33, 34] or to extend their functionality [16]. Davidson et al. [31] propose `FIE`, built on top of the `KLEE` symbolic execution engine, to incorporate new symbolic execution techniques. It can be used to verify security properties of some simple firmware images often found in practice. Zaddach et al. [70] describe Avatar, a dynamic analysis platform for firmware security testing. In Avatar, the instructions are executed in an emulator, while the IO accesses to the embedded system's peripherals are forwarded to the real device. This allows a security engineer to apply a wide range of advanced dynamic analysis techniques like tracing, tainting and symbolic execution.

A large set of firmware images of Xerox devices were reverse-engineered by Costin [24] leading to the discovery of hidden PostScript commands. Such commands allow an attacker to e.g., dump a device's memory, recover passwords, passively scan the network and more generically interact with devices' OS layers. Such attacks could be delivered to printers via web pages, applets, MS Word and other standard printed documents [23].

Bojinov et al. [18] conducted an assessment of the security of current embedded management interfaces. The study, conducted on real physical devices, found vulnerabilities in 21 devices from 16 different brands, including network switches, cameras, photo frames, and lights-out management modules. Along with these, a new class of vulnerabilities was discovered, namely *cross-channel scripting (XCS)* [17]. While XCS vulnerabilities are not particular to embedded devices, embedded devices are probably the most affected population. In a similar study, the authors manually analyzed ten Small Office/Home Office (SOHO) routers [48] and discovered at least two vulnerabilities per device.

Looking at insecure (remote) firmware updates, researchers reported the possibility to arbitrarily inject malware into the firmware of a printer [24, 27]. Chen [22] and Miller [58] presented techniques and implications of exploiting Apple firmware updates. In a similar direction, Basnight et al. [12] examined the vulnerability of PLCs to intentional firmware modifications. A general firmware analysis methodology is presented, and an experiment demonstrates how legitimate firmware can be updated on an Allen-Bradley ControlLogix L61 PLC. Zaddach et al. [72] explore the consequences of a backdoor injection into the firmware of a hard disk drive and uses it to exfiltrate data.

French and Casey [13] present fuzzy hashing techniques in applied malware analysis. Authors used `ssdeep` on *CERT Artifact Catalog* database containing 10.7*M* files. The study underlines the two fundamental challenges to operational usage of fuzzy hashing at scale: timeliness of results, and usefulness of results. To reduce the quadratic complexity of the comparisons, they propose assigning files into "bins" based on the block and file sizes. This approach, for their particular dataset and bins partitioning strategy, allowed for a given fuzzy hash to reduce the search space on average by 83.1%.

Finally, Bailey et al. [10] and Bayer et al. [14] propose efficient clustering approaches to identify and group malware samples at large scale. Authors perform dynamic analysis to obtain the execution traces of malware programs or obtain a description of malware behavior in terms of system state changes. These are then generalized into behavioral profiles which serve as input to an efficient clustering algorithm that allows authors to handle sample sets that are an order of magnitude larger than previous approaches. Unfortunately, this approach cannot be applied in our framework since dynamic analysis is unfeasible due to the heterogeneity of architectures used in firmware images.

8 Conclusion

In this paper we conducted a large-scale static analysis of embedded firmwares. We showed that a broader view on firmware is not only beneficial, but actually necessary

for discovering and analyzing vulnerabilities of embedded devices. Our study helps researchers and security analysts to put the security of particular devices in context, and allows them to see how known vulnerabilities that occur in one firmware reappear in the firmware of other manufacturers.

We plan to continue collecting new data and extend our analysis to all the firmware images we downloaded so far. Moreover, we want to extend our system with more sophisticated static analysis techniques that allow a more in-depth study of each firmware image. This approach shows a lot of potential and besides the few previously mentioned case studies it can lead to new interesting results such as the ones recently found by Costin et al. [25].

The summarized datasets are available at `http://firmware.re/usenixsec14`.

Acknowledgments

We thank the anonymous reviewers for their many suggestions for improving this paper. In particular we thank our shepherd, Cynthia Sturton, for her valuable time and inputs guiding this paper for publication. We also thank Pietro Michiardi and Daniele Venzano for providing access and support to their cloud infrastructure, and John Matherly of Shodan search engine for providing direct access to Shodan's data and resources.

The research leading to these results was partially funded by the European Union Seventh Framework Programme (contract Nr 257007 and project FP7-SEC-285477-CRISALIS).

References

[1] Audit PHP Configuration Security Toolkit.

[2] Define of backdoor string in DLink DI-524 UP GPL source code. `https://gist.github.com/ccpz/6960941`.

[3] Google Custom Search Engine API.

[4] Internet Census 2012 – Port scanning /0 using insecure embedded devices. `http://internetcensus2012.bitbucket.org`.

[5] SHODAN – Computer Search Engine. `http://www.shodanhq.com`.

[6] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[7] Slashdot: Backdoor found in TP-Link routers, March 2013.

[8] Download statistics for the wemo android application, February 2014. `http://xyo.net/android-app/wemo-JJUZgf8/`.

[9] Download statistics for the wemo iOS application, February 2014. `http://xyo.net/iphone-app/wemo-J1QNimE/`.

[10] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Z. Basnight, J. Butts, J. L. Jr., and T. Dube. Firmware modification attacks on programmable logic controllers. *International Journal of Critical Infrastructure Protection*, 6(2):76 – 84, 2013.

[13] L. Bass, N. Brown, G. M. Cahill, W. Casey, S. Chaki, C. Cohen, D. de Niz, D. French, A. Gurfinkel, R. Kazman, et al. Results of CMU SEI Line-Funded Exploratory New Starts Projects. 2012.

[14] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Symposium on Network and Distributed System Security*, NDSS '09. The Internet Society, 2009.

[15] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A View on Current Malware Behaviors. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association.

[16] A. Blanco and M. Eissler. One firmware to monitor'em all. *Ekoparty*, 2012.

[17] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: Cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 420–431, New York, NY, USA, 2009. ACM.

[18] H. Bojinov, E. Bursztein, E. Lovett, and D. Boneh. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA*, 2009.

[19] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011.

[20] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware

Analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[21] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[22] K. Chen. Reversing and exploiting an Apple firmware update. *BlackHat USA*, 2009.

[23] A. Costin. Hacking Printers for Fun and Profit.

[24] A. Costin. PostScript(um): You've Been Hacked.

[25] A. Costin and A. Francillon. Short Paper: A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, WiSec '14. ACM, 2014.

[26] A. Cui. Embedded Device Firmware Vulnerability Hunting with FRAK. *DefCon 20*, 2012.

[27] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Proceedings of the 20th Symposium on Network and Distributed System Security*, NDSS '13. The Internet Society, 2013.

[28] A. Cui, Y. Song, P. V. Prabhu, and S. J. Stolfo. Brave New World: Pervasive Insecurity of Embedded Network Devices. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 378–380, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] 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*, ACSAC '10, pages 97–106, New York, NY, USA, 2010. ACM.

[30] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 21st Symposium on Network and Distributed System Security*, NDSS '14. The Internet Society, 2014.

[31] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 463–478, Berkeley, CA, USA, 2013. USENIX Association.

[32] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[33] G. Delugré. Closer to metal: reverse-engineering the Broadcom NetExtreme's firmware. *Hack.lu*, 2010.

[34] L. Duflot, Y.-A. Perez, and B. Morin. What if You Can'T Trust Your Network Card? In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 378–397, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] K. Dunham. A fuzzy future in malware research. *The ISSA Journal*, 11(8):17–18, 2013.

[36] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS Certificate Ecosystem. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 291–304, New York, NY, USA, 2013. ACM.

[37] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 605–620, Berkeley, CA, USA, 2013. USENIX Association.

[38] B. Eshete, A. Villafiorita, and K. Weldemariam. Early Detection of Security Misconfiguration Vulnerabilities in Web Applications. In *Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security*, ARES '11, pages 169–174, Washington, DC, USA, 2011. IEEE Computer Society.

[39] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward Secure Embedded Web Interfaces. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[40] HDMoore. Security Flaws in Universal Plug and Play: Unplug, Don't Play, 2013.

[41] C. Heffner. binwalk – firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images.

[42] C. Heffner. littleblackbox – Database of private SSL/SSH keys for embedded devices.

[43] C. Heffner. Breaking SSL on Embedded Devices, December 2010.

[44] C. Heffner. Reverse Engineering a D-Link Backdoor, October 2013.

[45] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding Software License Violations Through Binary Code Clone Detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 63–72, New York, NY, USA, 2011. ACM.

[46] 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 Conference on Security Symposium*, Security'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[47] J. Hirsch and K. Bensinger. Toyota settles acceleration lawsuit after $3-million verdict. Los Angeles Times, October 25, 2013.

[48] Independent Security Evaluators. SOHO Network Equipment (Technical Report), 2013.

[49] IOActive. Critical DASDEC Digital Alert Systems (DAS) Vulnerabilities, June 2013.

[50] IOActive. stringfighter – Identify Backdoors in Firmware By Using Automatic String Analysis, May 2013.

[51] IOActive. Critical Belkin WeMo Home Automation Vulnerabilities, February 2014.

[52] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[53] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[54] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digit. Investig.*, 3:91–97, 2006.

[55] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental Security Analysis of a Modern Automobile. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society.

[56] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008.

[57] P. C. Messina, R. D. Williams, and G. C. Fox. *Parallel computing works !* Parallel processing scientific computing. Morgan Kaufmann, San Francisco, CA, 1994.

[58] C. Miller. Battery firmware hacking. *BlackHat USA*, 2011.

[59] Nvidia. CUDA – Compute Unified Device Architecture Programming Guide. 2007.

[60] OpenwallProject. John the Ripper password cracker. `http://www.openwall.com/john/`.

[61] OWASP. Top 10 Vulnerabilities, 2013.

[62] V. Roussev. Data Fingerprinting with Similarity Digests. In *IFIP Int. Conf. Digital Forensics*, pages 207–226, 2010.

[63] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 851–862, New York, NY, USA, 2013. ACM.

[64] S. Skorobogatov and C. Woods. Breakthrough silicon scanning discovers backdoor in military chip. In *Proceedings of the 14th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'12, pages 23–40, Berlin, Heidelberg, 2012. Springer-Verlag.

[65] J. V. Stough. distributed-python-for-scripting – DistributedPython for Easy Parallel Scripting.

[66] Tjaldur Software Governance Solutions. Binary Analysis Tool (BAT).

[67] A. Tridgell. rsync – utility that provides fast incremental file transfer.

[68] D. A. Wheeler. SLOCCount – a set of tools for counting physical Source Lines of Code (SLOC). `http://www.dwheeler.com/sloccount/`.

[69] xobs and bunnie. The Exploration and Exploitation of an SD Memory Card. *CCC – 30C3*, 2013.

[70] 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 21st Symposium on Network and Distributed System Security*, NDSS '14. The Internet Society, 2014.

[71] J. Zaddach and A. Costin. Embedded Devices Security and Firmware Reverse Engineering. *BlackHat USA*, 2013.

[72] J. Zaddach, A. Kurmus, D. Balzarotti, E.-O. Blass, A. Francillon, T. Goodspeed, M. Gupta, and I. Koltsidas. Implementation and Implications of a Stealth Hard-drive Backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 279–288, New York, NY, USA, 2013. ACM.