Jyväskylän yliopiston julkaisuarkisto
JYX
Jyväskylä University Digital Archive
UNIVERSITY OF JYVÄSKYLÄ

This is an electronic reprint of the original article.
This reprint *may differ* from the original in pagination and typographic detail.

Author(s): Costin, Andrei; Zarras, Apostolis; Francillon, Aurélien

Title: Towards Automated Classification of Firmware Images and Identification of Embedded Devices

Year: 2017

Version:

Please cite the original version:

Costin, A., Zarras, A., & Francillon, A. (2017). Towards Automated Classification of Firmware Images and Identification of Embedded Devices. In S. De Capitani di Vimercati, & F. Martinelli (Eds.), ICT Systems Security and Privacy Protection : 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings (pp. 233-247). Springer. IFIP Advances in Information and Communication Technology, 502. https://doi.org/10.1007/978-3-319-58469-0_16

All material supplied via JYX is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Towards Automated Classification of Firmware Images and Identification of Embedded Devices

Andrei Costin[1], Apostolis Zarras[2], and Aurélien Francillon[3]

[1] University of Jyväskylä, Finland
andrei.costin@jyu.fi
[2] Technical University of Munich, Germany
zarras@sec.in.tum.de
[3] EURECOM, France
aurelien.francillon@eurecom.fr

Abstract. Embedded systems, as opposed to traditional computers, bring an incredible diversity. The number of devices manufactured is constantly increasing and each has a dedicated software, commonly known as *firmware*. Full firmware images are often delivered as multiple releases, correcting bugs and vulnerabilities, or adding new features. Unfortunately, there is no centralized or standardized firmware distribution mechanism. It is therefore difficult to track which vendor or device a firmware package belongs to, or to identify which firmware version is used in deployed embedded devices. At the same time, discovering devices that run vulnerable firmware packages on public and private networks is crucial to the security of those networks. In this paper, we address these problems with two different, yet complementary approaches: firmware classification and embedded web interface fingerprinting. We use supervised Machine Learning on a database subset of real world firmware files. For this, we first tell apart firmware images from other kind of files and then we classify firmware images per vendor or device type. Next, we fingerprint embedded web interfaces of both physical and emulated devices. This allows recognition of web-enabled devices connected to the network. In some cases, this complementary approach allows to logically link web-enabled online devices with the corresponding firmware package that is running on the devices. Finally, we test the firmware classification approach on 215 images with an accuracy of 93.5%, and the device fingerprinting approach on 31 web interfaces with 89.4% accuracy.

1 Introduction

In the wake of the Internet of Things (IoT), embedded devices are becoming increasingly present in many computing and networked environments. In fact, multiple reports estimate an increase in the number of embedded devices in the next few years [16, 24]. These devices often rely on *network connectivity*, are administrated through *web interfaces*, and *firmware packages* are made available with new features and bug fixes. In addition, many firmware releases are available for each device leading to a large number of firmware images [12]; this number

will likely grow with the increasing number of newly deployed devices. Therefore, it is challenging to apply manual analysis, classification and fingerprinting, as it does not scale. Hence, novel, scalable, and automated approaches are needed.

Usually, a firmware image is custom made for a specific device. Although, it is relatively easy for a human to find the vendor, the version, and the device for which the firmware is intended, because embedded devices are very diverse, it is difficult to automatically link a device model and a firmware image without a learning system that supports them. At the same time, it is extremely hard for an automated system to categorize firmware files from unstructured download sites by device class or by vendor. While this can be automated for a few well-defined file categories, this becomes hard when crawling thousands of firmware images from a wide diversity of devices. Similarly, when administrating an embedded device, a human can have contextual clues about its firmware version, however, an automated system requires a different approach to device identification.

Within this context we formulate the following problems: (*i*) how to automatically label the brand and the model of the device for which the firmware is intended and (*ii*) how to automatically identify the vendor, the model, and the firmware version of an arbitrary web-enabled online device. File classification and (*web*) fingerprinting might seem trivial problems, however, such problems are not trivial and were addressed in different contexts, for file classification [5,26,27,32], device fingerprinting [6, 14, 19], and web fingerprinting [1, 2, 29, 33]. Moreover, these problems need to be addressed in a reliable and scalable manner which is independent of device, vendor, or custom protocols running on the device.

In this paper, we apply *Machine Learning* (ML) to classify firmware files according to their vendor or device type. First, we explore several feature sets derived from the characteristics of firmware images. Then, we recommend a feature set for this type of classification problems that we found to be optimal and show that our approach achieves high accuracy. Next, using sound statistical methods, such as confidence intervals, we estimate the performance of our classifiers for large scale datasets. Complementary to the previous approach, we build a fingerprinting database of web interfaces using emulated firmware images (similar to [11, 13]) and physical devices. We show that it is feasible to match an unknown embedded web interface to the list of known web fingerprints in our database by using multiple features such as the web interface sitemap or the HTTP protocol Finite-State Machine (FSM). Finally, we use multiple scoring systems to rank the web fingerprint matches. The outcomes reveal that we are able to accurately classify firmware and fingerprint embedded web interfaces.

In summary, we make the following main contributions:

- We are the first to apply ML in the context of firmware classification. For this we propose and study the firmware features and the ML algorithms that makes the classification effective, accurate, and feasible.
- We research the fingerprinting and identification of web-enabled embedded devices and their firmware version, and introduce fingerprinting features for the embedded web interfaces of physical and emulated devices.
- We present and discuss direct practical applications for both techniques.

2 Firmware Classification and Identification

In this section we show how we classify the firmware files at vendor or at device-type level. Specifically, we present the details of our classifier for which we use supervised ML. In supervised ML, the algorithms must be trained with a set of annotated (e.g., manually, computer-aided) samples before it can classify unknown or new samples. In our experiments we use *Decision Trees (DT)* and *Random Forests (RF)* algorithms that are able to handle better non-linear features, and are easier and faster to train. The supervised ML algorithms also require *features* that are used to partition and distinguish the learned classes of data. Feature selection is usually specific to the domain to which the ML is applied and thus it must be carefully performed and evaluated. Therefore, we first present a set of "naive" attempts and their limitations. Then, we present our dataset, the features we explore, and the motivations behind our selection. Finally, we measure the performance of our classifiers trained for firmware files.

2.1 Discussion on "Naive" Attempts

One "naive" attempt could be the use of the firmware filenames as the source of various information (e.g., vendor and device name, firmware version). In practice, there are several problems with such an attempt. First, there is no standard that specifies if and how the filenames should carry metadata information. In fact, many firmware images are released with generic names such as `firmware.bin` or `upgrade.fw`. Second, extracting information from filenames is domain specific and is non-trivial [4]. Third, often the filenames can be fake and not related to their content. This is a known problem in "free-riding" on P2P and file sharing networks [17]. It also constitutes a problem in malware and spam distribution, where a filename can be used to disguise the real function of the file [21]. Therefore, we consider the filenames to be an untrustworthy source of information, but it could *optionally* be used at later stages for cross-validation of the information.

Another "naive" attempt could be the compilation of a dictionary of hashes based on *all* firmware files. One could query this dictionary when trying to obtain information (e.g., vendor, product, version) for a previously obtained firmware image. Such an attempt could face several challenges. First, there is no database that provides a list of *all* the firmware images that were created and are available to date: firmware releases and updates are not standardized or never publicly released. Second, even if such a database would hypothetically exist and the hashes of all the firmware files to date would be known, the problem remains for the firmware released in the future. It would be hard, if not impossible, to classify future firmware releases with such an attempt. In fact, this is one of the main reasons why malware file classification techniques do not use it, and rather propose alternative ways to detect and classify malware samples [5, 32], including techniques based on ML [26, 27]. Finally, it could still be possible to use fuzzy hashing to classify unseen or future firmware images with the right firmware category (i.e., label) according to fuzzy hash similarity. However, fuzzy hashing has its own limitations and is not viable in practice.

2.2 Dataset

From a dataset of firmware images we collected over time, we select 215 images from 13 vendors that manufacture several type of devices [13]. We will refer to these vendors as *classification categories*. Each of these categories contains a varying number of firmware images. In fact, this is a realistic scenario since firmware release cycles and the numbers of released firmware are diverse and vary across vendors, and even across devices from the same vendor. Each classification category contains between 5 and 54 firmware images, with an average of 16 images per vendor. Finally, we create a special classification category of files for which we know that they are not firmware images. For example, such files include drivers and PDF or text documents, which are often released along with firmware updates at a common download location or in a common file archive [12].

2.3 Features Selection

The classification of a firmware file can be performed at vendor or at device-type level, depending on the granularity objectives. For consistency, we will refer to both vendor and device-type categorization as *classification categories*.

Firmware File Size. The file size of a full firmware upgrade for an embedded device is directly related to the hardware design and the functionalities of the device. At the same time, a firmware upgrade file cannot exceed the limited memory available in the particular device types which it targets. This motivates us to use *firmware file size* as a good feature to discriminate between firmware images of devices from different classification categories.

Firmware File Content. Most vendors use custom procedures to build and package their firmware upgrades. This makes the firmware images to have specific distribution and density of the information they contain. Therefore, we use information theory properties as features for ML. In this sense, we leverage the following characteristics of the firmware files as ML features: (*i*) file entropy (i.e., the informational density of bits per byte), (*ii*) arithmetic mean of file bytes, (*iii*) file compressibility percentage (i.e., an empirical value that is an upper bound of the Kolmogorov complexity), (*iv*) serial correlation value, (*v*) monte-carlo value and its estimation error, and (*vi*) chi-square distribution and its excess error. We will refer to the file entropy as *entropy* feature and to the rest of the features from the above list as the *entropy extended* features set.

Firmware File Strings. Many software packages, including firmware files, contain strings. These strings may embody copyright, debugging, or other information. They often also contain vendor or device specific information. Hence, the strings in a given firmware file represent a fingerprint of the corresponding firmware, device, and vendor. Consequently, the intersection of strings of each file within a particular classification category is a strong classification feature for that category. Suppose that an unknown firmware sample contains a string that is found within strings intersection of a *classification category A*. There are high chances that this sample is related to the files in the *classification category A*.

Unfortunately, many firmware files contain strings that are common across multiple classification categories. This may happen if the firmware uses common Free Open Source Software (FOSS) code such as Linux kernel or OpenSSL libraries. In this case, an unknown sample can match several different classification categories and can mislead the ML classifier. To overcome this, for each trained classification category we also build a dictionary that contains only strings *unique* to that category. Therefore, each classification category in the training set adds two different features: the *Category Strings Feature* (CSF) and the *Category Unique Strings Feature* (CUSF). Unfortunately, the CUSF feature derivation comes with a drawback which can limit the scalability of our techniques with larger datasets. Whenever a new firmware file is added to a given classification category, the entire CUSF process has to be re-run on the labeled dataset.

Fuzzy Hashing of Firmware File Content. Fuzzy hashing is a technique which provides the ability to compare two different items and determine a fundamental level of similarity between them. While the cryptographic hashing is used to determine if two different items are *identical*, the fuzzy hashing is used to decide if two different items are *homologous* (i.e., similar but not exactly the same). In our approach, we use *Context Triggered Piecewise Hashes* (CTPH) [20]. Intuitively, firmware files from a given classification category should be more "fuzzy hash similar" among themselves than cross-category. As such, for each trained classification category we build a list containing fuzzy hashes of files within the category. For a training or an unknown file, we compare its fuzzy hash with the fuzzy hashes in the list of each category. If there is at least one fuzzy hash match with similarity above an empiric threshold, the fuzzy hash feature of that category is set to 1; otherwise, is set to 0. Surprisingly, including the fuzzy hash similarities as features proved to result in worse classification accuracy as discussed in Section 2.4.

2.4 Evaluation

Running supervised ML experiments requires training sets. Since our dataset has the classification categories of varying lengths, we create the training sets by taking a constant percentage from each category as training samples. We start with 10% as *training set percentage* and then increment it by 10% until training set percentage reaches 90%. For each training set percentage, we run 100 experimental runs by randomly sampling the given percentage of files as training samples and running the training and classification. Finally, for each *training set percentage*, we compute its average classification accuracy and error based on results of each of the 100 experimental runs. For any experiment run, we use both the DT and RF algorithms so that we can compare their performance under various conditions. Since we use DT and RF algorithms, we do not perform cross-validation because these algorithms do it internally. The firmware classification performance for various ML algorithms, feature sets, and training sets size is summarized in the Figures 1, 2, 3, and 4. For each algorithm and features sets, the figures depict the average accuracy per 100 experimental runs for *training set size* increasing with 10% increments.

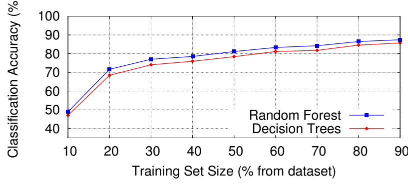

Fig. 1: Firmware classification performance using `[size, entropy]` features set.

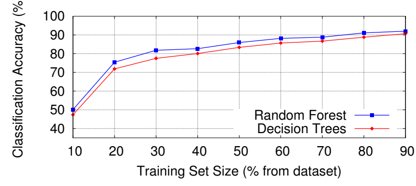

Fig. 2: Firmware classification performance using `[size, entropy, entropy extended]` features set.

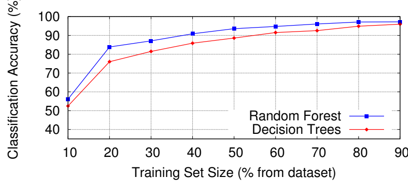

Fig. 3: Firmware classification performance using `[size, entropy, entropy extended, strings, strings unique]` features set.

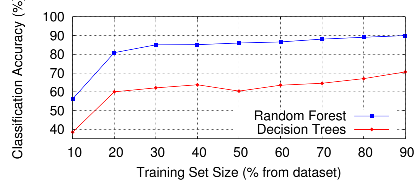

Fig. 4: Firmware classification performance using `[size, entropy, entropy extended, strings, strings unique, fuzzy hash]` features set.

First, we observe that the classification accuracy improves with the increased size of the training set. Although this appears to be trivial, is not always the case as there exist scenarios where larger datasets lead to worse results [30]. Second, contrary to the intuitive expectation, the addition of the fuzzy hash similarity features reduced the accuracy. Interestingly, these features made both the RF and the DT classifiers perform worse. The DT classifier also performed much worse compared to the DT classifiers with very basic feature sets, such as [`size`, `entropy`] or [`size`, `entropy`, `entropy extended`]. In parallel, the RF classifier in this setup failed to perform at least marginally better than the RF classifiers based on basic feature sets mentioned above. One explanation could be the fact that a fuzzy hash is not an accurate file match. Such hashing can return high similarity scores even for pair of files that are totally unrelated. The accuracy of the fuzzy hashing can be influenced by the file size and various other factors.

Based on the previous observations, we conclude that the feature set of [`size`, `entropy`, `entropy extended`, `category strings`, `category unique strings`] constitutes the best choice. It also provides best accuracy when used with the RF classifiers; more than 90% classification accuracy when the training set is based on at least 40% of the known firmware files. Another observation is that both the RF and the DT classifiers using other feature sets reach the 90% accuracy only for training set sizes of 80%–90% of the known firmware files, which is not practical in real-life. Also, the RF and the DT classifiers with the most basic feature set [`size`, `entropy`] does not even reach 90% classification.

We try to identify the most reliable feature set and learning classifier, however, the *generalization of learning* is an open problem in the ML field [7,8,15]. The ML algorithms performance cannot be guaranteed on another dataset (e.g., bigger). We compensate this limitation with statistical methods such as confidence intervals. In this context, we use statistical confidence intervals [10] to evaluate the accuracy of our technique when applied to real-world populations of firmware images. For example, by taking any firmware in a dataset of 172K firmware images [12], with an accuracy of 99% we can compute the confidence interval for our best feature set and a training based on 50% of the dataset. In this case, our RF firmware model can correctly classify the firmware in 93.5% $\pm$ 4.3% of the cases. Manually annotating 50% of a dataset with 172K firmware images is not trivial and does not scale. However, this challenge can be solved using alternative approaches. First, many files could be automatically annotated based on the metadata that was acquired by the crawler, assuming that the metadata from the vendor is reliable. Second, building in an incremental manner a clean training set can be achieved by using services like Amazon's Mechanical Turk.

3 Device Fingerprinting and Identification

Often firmware fingerprinting is not sufficient and thus it required to fingerprint the device itself. Many approaches exist for fingerprinting and identification of computing device and sensors [6, 14, 19]. However, the fingerprinting features used by these techniques are strongly linked to the real hardware or the way the live devices operate. Such strong dependencies can make these techniques less effective, for example, when dealing with emulated devices and virtualized appliances. In addition, these techniques do not necessarily take advantage of the devices' firmware packages. Often firmware packages can be emulated and can provide additional information for a reliable device identification.

At the same time, the embedded devices often lack the user interfaces of desktop computers, such as keyboard, video, and mouse. Nevertheless, these devices need somehow to be administrated. Even though some devices alternatively rely on custom protocols such as "thick" clients or even legacy interfaces (e.g., telnet), the web became the universal administration interface. Thus, the firmware of these devices often embed a web server providing a web interface and these web applications range from quite simple to fairly complex.

These observations suggest that higher level approaches are required, regardless the way the devices operate. As such, we propose an approach that fingerprints the devices at high level as possible, which in our case is the *embedded web interface* level. Our solution benefits from the firmware contents and the device emulation based on the firmware images alone. Previous works touched some aspects of our fingerprinting techniques, however, either they suggest manual approaches [22] or do not provide enough insights and evaluations [29]. The well recognized project such as ZMap/ZTag also include a device/service fingerprinting feature. However, their efforts in this regard have been mostly manual so far as seen in their GitHub and Travis-CI logs.

3.1 Discussion on "Naive" Attempts

One "naive" attempt to create or verify fingerprints of embedded web interfaces is to use physical devices in a private network or devices connected to the Internet with public IPs. In practice, there are several problems with such an attempt. First, it is unfeasible to operate physical devices in a private network for all the embedded devices that exist to date. Second, it could be unethical, or even sometimes illegal, to scan the devices connected to the Internet with public IPs for the purpose of analysis or fingerprinting without prior authorization. Another "naive" attempt could be to check the existence of unique files/URLs, or specific strings in the web interface pages or HTTP authentication prompts. However, in our view such an attempt cannot deal very well with false positives that can be produced by fake web pages created by web traffic generators, and fake services produced by rather simple honeypots [23]. Therefore, we suggest that more elaborate approaches must be designed and proposed.

3.2 Dataset

We used the emulated web interface of 27 firmware images originating from 3 vendors that split across 7 functional categories. 9 of these images where also part of the firmware ML classification: they were classified by our ML firmware model with an accuracy of 100% using RF (and around 99.5% using DT). There are practical reasons why we could not use the entire dataset of 215 images from the firmware classification experiment: (*i*) emulating a large number of diverse firmware images is a challenging problem [13] and (*ii*) it is unfeasible and expensive to acquire many devices such that their number is large enough to produce convincing and representative results. We also used 4 physical devices from 2 vendors that cover 4 functional categories. We consider that the dataset has a sufficient size and enough intramodel similarity to provide a conclusive estimation of the effectiveness and the accuracy of our technique.

3.3 Features for Web Interface Fingerprinting

We propose six different features that are computed for each training or unknown embedded web interface, which present them below and motivate the choice.

Web Sitemap. A sitemap is a list of pages of a website which are publicly or privately accessible. Files and URLs that exist in one website do not necessarily exist in another, even if they run on the same web server. We leverage this fact and create a fingerprint based on this assumption. In detail, to categorize the web interface of an unknown embedded device, we access URLs and files which exist in our trained dataset and represent the sitemap of a known web application. If the sitemap of the unknown web interface matches with a known one in our database, we classify it as belonging to an embedded device running a specific firmware version. This sitemap approach however would not work for single-page web applications that use JavaScript router scripts. This could be addressed by fingerprinting the Document Object Model (DOM) of those interfaces.

HTTP Finite-State Machine. The HTTP protocol, is a stateless application-level protocol for distributed, collaborative, and hypermedia information systems. For our fingerprinting and detection purposes we focus only on the server responses. HTTP is a liberal protocol which means that the structure of a response message is diversified among the different web server implementations. Each web server implements the response messages differently in terms of the headers it uses, the sequence of these headers inside the message, and the value of each header. Hence, it is possible to fingerprint them by extensively analyzing the messages they exchange [31,34]. We leverage these differences to identify the type of server involved in a specific HTTP conversation. Therefore, we create a model which is able to learn the headers' order of an HTTP response and then use this order to classify an unknown HTTP conversation. In essence, we have implemented an HTTP FSM in which the headers represent the states of this FSM and the order of the headers the transitions from one state to another.

Cryptographic Hashing of HTML Content and HTTP Headers. We expand the FSM approach by using also the HTTP response values. Although some headers will always display the same information (e.g., the header *Server*) few other headers will not remain constant over time (e.g., the header *Date*). Such small variations in responses results in significant changes in the cryptographic hashes of the headers. For example, the cryptographic hashes of headers of two consecutive responses to exactly the same static web resource will result in two different values and will generate a false mismatch. To overcome this type of "noise", instead of retrieving the actual value of such a header, we dynamically create a regular expression. As a consequence, headers such as *Date* do not affect our features and matching. We create two cryptographic hash values from a complete HTTP response. The first contains the hashed headers of the message as explained above and the second the hashed message body. If we have a match between an unknown response and a response contained in our database, we can successfully fingerprint the device that sent this response. However, many times an HTTP response from an unknown device will match a list of devices that can reply back with responses that hash to the same values. In those cases, we can use this approach to minimize the number of devices that match this response.

Fuzzy Hashing of HTML Content and HTTP Headers. It is not always possible to have a fingerprint based on a cryptographic hash value even if it comes from the same device. This happens because the modification of a single byte in a large byte stream causes the cryptographic hash function to generate a completely different hash value. To counter this behavior we use fuzzy hashing. In our approach we use *Context Triggered Piecewise Hashes* (CTPH) [20]. Using fuzzy hashing, we compute the similarity between an unknown HTTP response (HTTP headers, HTML content) and a list of HTTP responses in our database for which we know the fuzzy hash values. The procedure we follow is quite similar to the one we followed in the case of the cryptographic hashing, but in this approach we are using a completely different hashing function. If the similarity between the unknown and a known HTTP response exceeds an empirically calculated threshold, we can successfully classify this unknown device.

3.4 Scoring Systems for Features

Scoring is the way each feature contributes to the final rank of a given match. We propose three different scoring systems and briefly present them below.

Majority Vote Scoring. Each feature of each fingerprint match is ranked in decreasing order. The fingerprint match that ranks highest on most of its features is considered to be the most accurate match to the unknown sample.

Uniform and Non-Uniform Weighs Scoring. Each feature value of a fingerprint is assigned a weight. Then, for each feature of each fingerprint all the weighted values are summed into a total value of the fingerprint. Finally, all the total values are ranked in decreasing order. The match whose total value ranks highest is considered as the most accurate match to the unknown sample. For our evaluation, we used the uniform weights of 16.6% (i.e., a uniform devision of 100% match to each of the six features). For the non-uniform scoring, we used the following empirically found weights: 4% for `web sitemap`, 4% for `HTTP FSM`, 1% for `fuzzy hash HTTP header`, 1% for `fuzzy hash HTML content`, 10% for `crypto hash HTTP header`, 80% for `crypto hash HTML content`.

Score Fusion. In our evaluation, we used the *score fusion* technique to improve the accuracy of identification. The score fusion technique is widely and actively used in various research fields, such as biometrics and sensors data [18]. It is used to increase the confidence in the results and to counter the effect of imprecisely approximated data (e.g., fingerprints in biometrics) and unstable data readings (e.g., sensors data). We take as input the decreasingly ordered rankings from each of the scoring systems described above. Then, we apply majority voting to each ranking from these three scoring systems. This allows our system to decide which match is the most accurate based on its scores computed using the three different scoring systems presented above.

3.5 Evaluation

We start by connecting up the 4 physical devices and emulating the 27 firmware images. Then we create one fingerprint for the embedded web interface of each of these 31 devices. Subsequently we create a list of IP addresses based on the IP address of each of the 31 running devices. We feed sequentially each IP address to the identification module which acts like an *oracle* and has to "guess" to what fingerprint to assign the web-interface at this particular IP address. For this, the identification module loads the previously created fingerprinting database, computes the features' scores for each URL and accumulates them, runs the scoring systems and finally outputs the most accurate fingerprint match by applying the score fusion method. The list of these steps constitutes an experimental run. We execute the above steps for 100 experimental runs at various points in time, under varying network conditions and varying IP address assignments. We also vary the number of threads used for web interface crawling and the speed at which they crawl. Finally, we compute the average successful and erroneous identification rates based on results from each experimental run.

Summarized, our tests on average resulted in 89.4% accuracy in device identification. The tests were run using a database containing 31 fingerprints of embedded web interfaces. Also, the dataset provided enough intramodel similarity, because around a third of the fingerprinted web interfaces consist of similar or consecutive firmware versions (e.g., Brickcom). Our evaluations show that the cryptographic hash of the content is the most stable and accurate feature. On average, it provided an accuracy of more than 85%. On the other end, the fuzzy hash of headers and content were the most unstable. One reasons for this is that fuzzy hashing does not perform well with short data (e.g., HTTP headers). Another reason, as discussed also in Section 2.4, could be the fact that fuzzy hash is not an accurate data match and can introduce noise rather than useful similarity information. These empirical observations lead us to choose the non-uniform scoring weights as presented in Section 3.4. Finally, the most accurate scoring system in our tests was the majority voting, followed by the non-uniform. As expected, the uniform weights scoring system performed the worst with more than 50% of classification errors. This could be explained by the high weight values assigned to the fuzzy hash features which can be noisy and inaccurate.

4 Usage Scenarios

While taking a research-oriented approach to the open problems, with this work we also aim at providing practical results and usability. Thus, we consider that providing real-life examples and applications is equally important.

4.1 Firmware Classification

Correct identification and classification of a firmware could be extremely useful. First, it could allow easy and fast clustering and navigation of firmware files according to their vendor or device type. Subsequently, this more granular separation could be used to apply more refined techniques on each category, such as version or release date ordering within the category. With all the files in one huge cluster, such refined techniques would be more challenging, if not impossible. Second, it could allow to build a database of firmware images associated with inputs that trigger their vulnerabilities. Finding such vulnerabilities is an interesting topic itself, but is outside the scope of this paper and were addressed in [12, 13]. Subsequently, for new firmware releases labeled into a category, the vulnerability triggering inputs from older firmware releases could be automatically tested. This could be used to test if a specific vulnerability was fixed in the last version or it is still present. Third, for well classified firmware files only a specific set of firmware effective unpackers are run. Unpacking firmware files is known to be resource and time consuming [12]. Applying a specific set of effective unpackers skips the brute force unpacking, thus saving processing time and providing faster and more accurate results. Finally, once the vendor is known, as a result of a successful firmware classification, vendor-specific analysis techniques and tools may be applied. The specifics can be tuned based on the knowledge of vendor's development practices and technologies used.

4.2 Device Fingerprinting and Identification

Defensive use. Our technique may be used to scan a network and fingerprint the detected embedded web interfaces. The fingerprint information may be used to identify the device model and vendor, and its firmware version. This information can also be used to offer a firmware upgrade if the identified firmware version running on the device is obsolete. The remaining unidentified devices in the network could be easily annotated by the user with attributes such as vendor, device model, and firmware version.

Offensive use. A penetration tester performing a black-box test may use our device fingerprinting and identification technique to identify the device model and the firmware version of an unknown embedded device encountered during the test. With this information, CVEs or exploits could be automatically retrieved for the particular device model and firmware version. This may help increase the test's success rate and decrease the required test time. Recent evidence shows that a similar but simplified technique was used for offensive use by a malvertising campaign targeting home routers and similar embedded devices [25].

4.3 Automated End-to-End Scenario

It is practical to have a system that can address the *firmware vulnerability discovery* and the *vulnerable device discovery* in an end-to-end autonomous process. The proposed techniques are an ideal complement for the firmware vulnerability discovery techniques [11–13]. First, the crawlers collect firmware images. Then, the ML techniques from Section 2 classify the firmware. Using static and dynamic analysis as well as emulation techniques on both generic and firmware-specific processing, we can discover vulnerabilities within the firmware. Once the emulator of the firmware is functional, the techniques from Section 3 create a fingerprint for the emulated device and firmware version. Finally, using scanning tools such as Shodan.io and Censys.io (Internet), or nmap and Nessus (Intranet), we can identify devices based on their fingerprint, and immediately label and isolate them according to discovered vulnerabilities. This way, we can cover the entire vulnerability life-cycle in an automated manner.

5 Related Work

Kohno et al. [19] introduced the area of remote physical device fingerprinting. Desmond et al. [14] proposed a fingerprinting technique that differentiates between unique devices through timing analysis of 802.11 probe request frames. Shah [29] presented early techniques to fingerprint and identify web applications at the HTTP level. Similar, the BlindElephant [1] attempts to discover the version of a web application. While, Wapplyzer [33] uses regular expressions to uncover the technologies used on websites, WhatWeb [2] uses more than 900 plugins to recognize the web technologies used within a website. Alvarez [3] used the

Extended File Information (EXIF) metadata in JPEG files to generate fingerprints, and Bongard [9] studied the implementation differences among the PNG codecs used with the most popular web application platforms. Samarasinghe and Mannan [28] used TLS/SSL certificate details to fingerprint embedded devices.

6 Conclusion

In this paper we presented two complementary techniques, *embedded firmware trained classification* and *embedded web interface fingerprinted identification*. We proposed ML for the firmware classification challenge, and explored features and score fusion to address the web interface fingerprinting and identification problem. With high confidence for real-world large scale datasets, our tests demonstrated that the classifiers and features we proposed achieved 93.5% accuracy in firmware classification and 89.4% accuracy in device identification. Finally, we presented practical use cases of our techniques which motivated our work. The datasets and scripts will be available at `http://firmware.re` project page.

Acknowledgments

The research was partially supported by the German Federal Ministry of Education and Research under grant 16KIS0327 (IUNO).

References

1. BlindElephant Web-App Fingerprint. `http://blindelephant.sourceforge.net`.
2. WhatWeb. `http://morningstarsecurity.com/research/whatweb`.
3. P. Alvarez. Using Extended File Information (EXIF) File Headers in Digital Evidence Analysis. *International Journal of Digital Evidence*, 2(3):1–5, 2004.
4. N. Anquetil and T. Lethbridge. Extracting concepts from file names: a new file clustering criterion. In *International Conference on Software Engineering*, 1998.
5. M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2007.
6. A. Bates, R. Leonard, H. Pruse, D. Lowd, and K. Butler. Leveraging USB to Establish Host Identity Using Commodity Devices. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.
7. C. M. Bishop. Pattern Recognition and Machine Learning. *Machine Learning*, 2006.
8. A. L. Blum and P. Langley. Selection of Relevant Features and Examples in Machine Learning. *Artificial intelligence*, 97(1):245–271, 1997.
9. D. Bongard. Fingerprinting Web Application Platforms by Variations in PNG Implementations. *Blackhat*, 2014.
10. J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, 2011.
11. D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

12. A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, 2014.
13. A. Costin, A. Zarras, and A. Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.
14. L. C. C. Desmond, C. C. Yuan, T. C. Pheng, and R. S. Lee. Identifying Unique Devices Through Wireless Fingerprinting. In *Wireless Network Security*, 2008.
15. P. Domingos. A Few Useful Things to Know About Machine Learning. *Communications of the ACM*, 55(10):78–87, 2012.
16. Intel. Rise of the Embedded Internet, 2009.
17. M. Karakaya, I. Korpeoglu, and Ö. Ulusoy. Free Riding in Peer-To-Peer Networks. *IEEE Internet Computing*, 13(2):92–98, 2009.
18. L. A. Klein. *Sensor and Data Fusion: A Tool for Information Assessment and Decision Making*. SPIE Press Bellingham, 2004.
19. T. Kohno, A. Broido, and K. C. Claffy. Remote Physical Device Fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005.
20. J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digital investigation*, 3:91–97, 2006.
21. F. Li, A. Lai, and D. Ddl. Evidence of Advanced Persistent Threat: A Case Study of Malware for Political Espionage. In *International Conference on Malicious and Unwanted Software (MALWARE)*, 2011.
22. M. Niemietz and J. Schwenk. Owning Your Home Network: Router Security Revisited. In *Web 2.0 Security and Privacy (W2SP)*, 2015.
23. Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. IoTPOT: analysing the rise of IoT compromises. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.
24. Postscapes. Internet of Things Market Forecast, 2014.
25. Proofpoint Inc. Home Routers Under Attack via Malvertising on Windows, Android Devices. `https://www.proofpoint.com/us/threat-insight/post/home-routers-under-attack-malvertising-windows-android-devices`.
26. K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and Classification of Malware Behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.
27. J. Sahs and L. Khan. A Machine Learning Approach to Android Malware Detection. In *European Intelligence and Security Informatics Conference*, 2012.
28. N. Samarasinghe and M. Mannan. Short Paper: TLS Ecosystems in Networked Devices vs. Web Servers. In *International Conference on Financial Cryptography and Data Security (FC)*, 2017.
29. S. Shah. HTTP Fingerprinting Advanced Assessment Techniques. *Blackhat*, 2003.
30. S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.
31. G. Stringhini, M. Egele, A. Zarras, T. Holz, C. Kruegel, and G. Vigna. B@bel: Leveraging Email Delivery for Spam Mitigation. In *Usenix Security*, 2012.
32. R. Tian, L. M. Batten, and S. Versteeg. Function Length as a Tool for Malware Classification. In *Conference on Malicious and Unwanted Software*, 2008.
33. Wappalyzer. Identify Technology on Websites. `https://wappalyzer.com`.
34. A. Zarras, A. Papadogiannakis, R. Gawlik, and T. Holz. Automated Generation of Models for Fast and Precise Detection of HTTP-based Malware. In *International Conference on Privacy, Security and Trust (PST)*, 2014.