ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0055

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Andrei COSTIN

le 23 Septembre 2015

Analyse à large échelle de la sécurité du logiciel dans les systèmes embarqués

Directeur de thèse : **Aurélien FRANCILLON**
Co-encadrement de la thèse : **Davide BALZAROTTI**

Jury

M. Srđan ČAPKUN, Professeur, ETH, Zürich, Suisse	Rapporteur
M. Claude CASTELLUCCIA, Senior Research Scientist, INRIA, Grenoble, France	Rapporteur
M. Renaud PACALET, Directeur d'Études, Télécom ParisTech, Sophia-Antipolis, France	Examinateur
Mme. Nathalie MESSINA, SW Architect, Magneti Marelli, Sophia-Antipolis, France	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

T H È S E

Large Scale Security Analysis of Embedded Devices' Firmware

Thesis

Andrei Costin
andrei.costin@eurecom.fr

École Doctorale Informatique, Télécommunication et Électronique, Paris
ED 130

Publicly defended on: September 23rd, 2015

Advisor:
Assistant Prof. Aurélien Francillon
EURECOM, Sophia-Antipolis

Co-Advisor:
Assistant Prof. Davide Balzarotti
EURECOM, Sophia-Antipolis

Reviewers:
Prof. Srđan Čapkun,
ETH, Zurich

Dr. Claude Castelluccia,
INRIA, Grenoble

Examiners:
Dr. Renaud Pacalet,
Télécom ParisTech, Sophia-Antipolis

Dr. Nathalie Messina,
Magneti Marelli, Villeneuve-Loubet

This thesis is dedicated to my wife and my parents.
For their love, support and encouragement.

Acknowledgements

I want to start by thanking my supervisors, Aurélien and Davide. They were a true inspiration and strong motivators throughout the uneasy road towards a PhD thesis and title. They were always there when I was looking for an advise, and they knew how to help when apparently there was nothing more to be done. I was very fortunate and blessed that I chose them and, more importantly, that they chose me for this PhD. Thank you for your trust and support, and for being there (remotely, or close-by with pizza and beer) during the hectic paper submissions just seconds before late AM hours of CFP deadlines.

I also thank Aurélien and Davide for their help organizing my thesis defense committee (spoiler alert: much more challenging than it sounds!), whose members I thank for their time and effort allotted for reviewing my work and for their constructive questions and feedback during the entire process.

My regards also go to the S3 SysSec group and its former, present and future members. In particular, my best go to: Jonas, Mariano, Davide C., Leyla, Merve and Onur, Luca, Sandeep, and Jelena. The S3 SysSec group would not be what it is without its collaborating groups and visiting friends: many thanks go to Apostolis for the nice and fruitful collaborations we've been developing since his visit at S3 SysSec; and to Lucian for the after-work hours spent with the Romanians' group over good beers during his S3 SysSec experience exchange.

A special thanks goes to our very close friends: Igor, Misca and Nastea, Slava and the Moiseev family, the Burlacu family, the Ciur-Panfilov family, the Bogulean family. My best also go to the friends of the STAN group, and some more shout-outs go to our Romanian friends in the Sophia-Antipolis area. Their support, warm welcoming to their homes and fun discussions about the past memories, the present joys and the future made my PhD time much more fun and enjoyable, and also helped me move my PhD research forward.

Moving closer to the thesis' submission and defense brings a lot of pressure, stress and uncertainty. What I have learned is that sometimes having the opportunity to disconnect from everything can be the key. In this regard, I was fortunate to have been part of (read: to have escaped to) very nice two summer schools (yes, exactly like in the high-school!) which helped me disconnect, relax, and focus in silence on the thesis and on important things in life. First, my thanks go to UCLA's IPAM (and their awesome staff) and to Aurélien for giving me the opportunity to spend three great weeks in Los Angeles as part of the GSS2015 program. Second, my thanks go to COINS Research School of Computer and Information Security (and their helpful staff), and in particular to Hanno Langweg. It was that kind of "last-minute call" experience when by a total surprise

you get invited to give a lecture, with the added bonus of enjoying an amazing summer week in Metochi on the beautiful Greek island of Lesbos. I found the Metochi monastery extension a very special place and being there was a great opportunity for me to meditate during the silent and pleasantly warm evenings, to find my inner balance, and to calmly move the thesis to its logical and timely conclusion. My regards go to the new friends I've met during these two summer schools and with whom we've spent great summer time both in Los Angeles and Lesbos!

Thank you all!

Last, but not least, no matter how hard I try, it will never be enough to say $thank\ you$ to my wife and to my family. Their love, trust and support is at the core of this thesis and the cornerstone of my PhD title. If the PhD experience was challenging for me to say the least, I am more than confident it was not quite easy for them either. I made them endure a lot, you name it: sleepless nights, endless deadlines, numerous dry-runs, countless "just one more experiment and I call you back" moments, disappearing for "just (yet) another conference" travel, requests for reviewing and proof-reading my papers, mood swings, burnouts, and the list can go forever. And they never turned their back on me. Instead, they kept walking right along with me and gave me the best of their advise and support. Their endless patience and love was and is key to my success

– THANK YOU AND I LOVE YOU!

Abstract

Embedded systems are omnipresent in our everyday life and are becoming increasingly present in many computing and networked environments. For example, they are at the core of various Common-Off-The-Shelf (COTS) devices such as printers, video surveillance systems, home routers and virtually anything we informally call $electronics$. The emerging phenomenon of the Internet-of-Things (IoT) will make them even more widespread and interconnected. Cisco famously predicted that there will be 50 billion connected embedded devices by 2020. Given those estimations, the heterogeneity of technology and application fields, and the current threat landscape, the security of all those devices becomes of paramount importance. In addition to this, manual security analysis does not scale. Therefore, novel, scalable and automated approaches are needed.

In this thesis, we present several methods that make feasible the large scale security analysis of embedded devices. We implemented those techniques in a scalable framework that we tested on real world data. First, we collected a large number of firmware images from Internet repositories. Then we unpacked a large subset of them and performed simple static analysis. This resulted in the discovery of many new vulnerabilities. Also, this allowed us to identify five important challenges.

Embedded devices often expose web interfaces for remote administration. Therefore, we developed techniques for large scale static and dynamic analysis of such interfaces. This allowed us to find a large number of new vulnerabilities and to identify the limitations of emulation and web security tools.

Finally, identifying and classifying the firmware files is difficult, especially at large scale. For these reasons, we proposed Machine Learning (ML) techniques and features for firmware files classification. Also, we developed multi-metric score fusion approaches to fingerprint and identify embedded devices at the web interface level.

Using these techniques, we were able to discover a large number of new vulnerabilities in dozens of firmware packages, affecting a great variety of vendors and device classes. We were also able to achieve high accuracy in fingerprinting and classification of both firmware images and live embedded devices.

Résumé en français

Les systèmes embarqués (embedded systems) sont omniprésents dans notre vie quotidienne et sont de plus en de plus présents dans nombreux environnements informatiques en réseau. Par exemple, ils sont à la base de divers dispositifs Common-Off-The-Shelf (COTS) tels que les imprimantes, les routeurs et pratiquement tout ce que nous appelons communément les appareils électroniques. Le phénomène émergent de l'Internet des objets (Internet-of-Things (IoT)) va rendre ces systèmes encore plus communs et interconnectés. Cisco prévoit qu'il ait 50 milliards de systèmes embarqués connectés en 2020. Compte tenu de ces estimations, de l'hétérogénéité des domaines technologiques et d'application, et des menaces potentielles, la sécurité de tous ces dispositifs est d'une importance primordiale. De plus, une analyse manuelle de leur sécurité ne passe pas à l'échelle. Donc, de nouvelles approches automatisées et qui passent à l'échelle sont donc nécessaires.

Dans cette thèse, nous présentons plusieurs méthodes qui rendent possible l'analyse de la sécurité des dispositifs embarqués à grande échelle. Nous avons implémenté ces techniques dans un système évolutif que nous avons testé sur les données de systèmes réels. Tout d'abord, nous avons recueilli un grand nombre d'images logicielles (firmware) sur Internet. Ensuite, nous avons dépaqueté un grand nombre d'entre eux et nous avons réalisé une analyse statique simple. Cela a permis de découvrir beaucoup de nouvelles vulnérabilités ainsi que d'identifier cinq défis de recherche importants. Souvent, les appareils embarqués exposent des interfaces Web pour l'administration à distance. Par conséquent, nous avons développé des techniques pour l'analyse statique et dynamique de ces interfaces à grande échelle. Cela nous a permis de trouver un grand nombre de nouvelles vulnérabilités et d'identifier les limites de l'émulation et les limites des outils d'analyse de la sécurité Web. Enfin, il est difficile d'identifier et de classer les firmwares, surtout à grande échelle. Pour ces raisons, nous avons proposé des techniques de Machine Learning et les caractéristiques discriminantes pour classer les firmwares. Aussi, nous avons développé des approches d'empreintes numériques et d'identifier dispositifs embarqués au à partir de leur interface Web.

Grâce à ces techniques, nous avons découvert un grand nombre de nouvelles vulnérabilités dans de nombreux firmwares, affectant une grande variété de les fournisseurs et les classes de périphériques. Nous avons également été en mesure d'atteindre une grande précision dans les empreintes numériques et la classification des firmwares et les dispositifs embarqués.

Rezumat în română

Dispozitivele încorporate sunt omniprezente în viața noastră de zi cu zi și devin din ce în din ce în ce prezente în multe medii de calcul și de rețea. De exemplu, ele sunt la baza diverselor dispozitive de uz comun (Common-Off-The-Shelf (COTS)), cum ar fi imprimantele, sistemelele de supraveghere video, routerele și practic orice noi numim convențional $electronice$. Fenomenul emergent "Internetul Obiectelor" (Internet-of-Things (IoT)) le va face chiar mai răspândite și interconectate. Cisco estimează că până în anul 2020 vor fi 50 de miliarde de dispozitive integrate conectate. Având în vedere aceste estimări, eterogenitatea domeniilor tehnologice și de aplicare, și peisajul actual al amenințărilor de securitate și de atacuri, securitatea tuturor acelor dispozitive devine de o importanță majoră. În afară de aceasta, este cunoscut faptul că analiza manuală de securitate nu este scalabilă. Prin urmare, sunt necesare abordări care sunt noi, scalabile și automatizate.

În această teză, vom prezenta mai multe metode care fac posibilă analiza de securitate la scară largă a dispozitivelor încorporate. Am implementat aceste tehnici într-un sistem automatizat și scalabil pe care le-am testat apoi pe date reale. Mai întâi, am colectat un număr mare de fișiere de firmware de pe Internet. Apoi am despachetat un subset mare din acele fișiere de firmware si am realizat o analiză statică simplă. Acest lucru a dus la descoperirea a numeroase vulnerabilități noi. De asemenea, acest lucru ne-a permis identificarea a cinci probleme principale și fundamentale asociate cu astfel de cercetări.

Dispozitive integrate expun adesea interfețe web pentru administrarea de la distanță. Prin urmare, am dezvoltat tehnici de analiză statică și dinamică la scară largă a unor astfel de interfețe. Acest lucru ne-a permis să găsim un număr mare de noi vulnerabilități și a identificăm limitările instrumentelor de emulare și de securitate web.

În cele din urmă, identificarea și clasificarea fișierelor firmware este dificilă, mai ales la scară largă. Din aceste motive, am propus tehnici de Machine Learning (ML) și caracteristici de clasificare pentru fișierele firmware. De asemenea, am dezvoltat abordări de fuziune a scorurilor multi-metrice pentru amprentarea digitală și identificarea dispozitivelor încorporate la nivelul interfeței web.

Folosind aceste tehnici, am reușit să descoperim un număr mare de noi vulnerabilități în zeci de fișiere firmware, care afectează o mare varietate de furnizori și clase de dispozitive. De asemenea, am reușit să atingem o acuratețe ridicată în amprentarea și clasificarea fișierelor firmware și a dispozitivelor integrate on-line.

Аннотация на русском

Встроенные системы (embedded systems) являются вездесущими в нашей повседневной жизни и становятся все более распространенными во многих вычислительных и сетевых средах. Например, они находятся в основе различных стандартных устройств (Common-Off-The-Shelf (COTS)), таких как принтеры, системы видеонаблюдения, маршрутизаторы и практически все, что мы неофициально называем *электроникой*. Новая технология "Интернет Вещей" (Internet-of-Things (IoT)) даст им еще большее распространение и сделает их более взаимосвязаными. Cisco как известно предсказал, что к 2020 году во всем мире будет насчитываться около 50 миллиардов встраиваемых устройств, которые будут подключены к Интернету. Учитывая эти оценки, неоднородность технологий и приложений, и текущий пейзаж угроз и атак, безопасность этих устройств станет первостепенной важности. Также известно что ручной анализ не масштабируется. Соответственно, существует потребность в новых, масштабируемых и автоматизированных подходах.

В данной диссертации, мы представляем несколько методов, которые делают возможным крупномасштабный анализ безопасности встраиваемых устройств. Мы реализовали полностью автоматизированную систему, внедрили в неё данные методы методы и проверили нашу систему на реальных данных. Во-первых, мы собрали большое количество файлов прошивки (firmware) из хранилищ Интернета. Затем мы распаковали большое количество из них и провели простой статический анализ. Это привело к обнаружению многих новых уязвимостей. Кроме того, это позволило нам определить пять важных проблем, которые являются специфическими для таких исследований. Встроенные устройства часто отркывают веб-интерфейсы для удаленного администрирования. Таким образом, мы разработали методы для крупномасштабного статического и динамического анализа таких интерфейсов. Это позволило нам найти большое количество новых уязвимостей, и определить ограничения эмуляции и инструментов тестирования веб-безопасности. Наконец, идентификация и классификация прошивок является сложной задачей, особенно в крупномасштабных установках. По этим причинам, мы предложили методы машинного обучения (ML), а также характеристики для классификации прошивок. Кроме того, мы разработали подходы для слияния много-метрических показателей для сбора цифровых “отпечатков” и идентификации встроенных устройств.

Используя эти методы, мы смогли обнаружить большое количество новых уязвимостей во многих прошивках. Эти уязвимости затрагивают большое разнообразие поставщиков устройств и классов встраиваемых устройств. Мы также смогли добиться высокой точности в сборе отпечатков и классификации прошивок и подключенных встроенных устройств.

Contents

List of Publications (2012—2015)

This dissertation is based on several papers that I have contributed to during my PhD.

The motivating example in Chapter 3 is based on the ACM WiSec 2014 *"SHORT PAPER: A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design"* paper.

The large scale security analysis of firmware images in Chapter 4 is based on the USENIX Security 2014 *"A Large Scale Analysis of the Security of Embedded Firmwares"* paper.

The Chapter 5, and Chapter 6 respectively, are based on the *"Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces"* paper, and the *"Scalable Firmware Classification and Identification of Embedded Devices"* paper respectively. At the time of the thesis submission, both of these papers are $under\ submission$ at ISOC NDSS'16.

Following is a complete list of papers that I have contributed to during my PhD studies. The papers marked with † are not included in this thesis (listed for completeness).

Conference and Journal Publications, Short Papers

1. **A. Costin**, A. Zarras, A. Francillon, *"Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces"*, *submitted at* Annual Symposium on Network and Distributed System Security (NDSS'16), August 2015 [80].

2. **A. Costin**, A. Zarras, A. Francillon, *"Scalable Firmware Classification and Identification of Embedded Devices"*, *submitted at* Annual Symposium on Network and Distributed System Security (NDSS'16), August 2015.

3. **A. Costin**, J. Zaddach, A. Francillon, D. Balzarotti, *"A Large Scale Analysis of the Security of Embedded Firmwares"*, Proceedings of the 23rd USENIX Security Symposium (USENIX Security), San Diego USA, August 2014 [79].

4. **A. Costin**, A. Francillon, *"SHORT PAPER: A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design"*, Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (ACM WiSec), Oxford UK, July 2014 [77].

5. † **A. Costin**, A. Francillon, *"Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices"*, BlackHat, Las Vegas USA, August 2012 [76].

6. † **A. Costin**, *"All your cluster-grids are belong to us: Monitoring the (in)security of infrastructure monitoring systems"*, Proceedings of the 1st IEEE Workshop on Security and Privacy in the Cloud (SPC) (co-located with IEEE Conference on Communications and Network Security (CNS)), Florence Italy, September 2015 [75].

7. † **A. Costin**, J. Isachenkova, M. Balduzzi, A. Francillon, D. Balzarotti, *"The Role of Phone Numbers in Understanding Cyber-Crime Schemes"*, Proceedings of the Annual Conference on Privacy, Security, and Trust (PST), Tarragona Spain, July 2013 [78].

8. † J. Isacenkova, O. Thonnard, **A. Costin**, D. Balzarotti, A. Francillon, *"Inside the SCAM Jungle: A Closer Look at 419 Scam Email Operations"*, Proceedings of the International Workshop on Cyber Crime (co-located with IEEE Symposium on Security and Privacy (SP)), San Francisco USA, 2013 [137].

9. † J. Isacenkova, O. Thonnard, **A. Costin**, D. Balzarotti, A. Francillon, *"Inside the SCAM Jungle: A Closer Look at 419 Scam Email Operations"*, EURASIP Journal on Information Security, 2014.

Invited Talks, Paper and Poster Presentations

1. **A. Costin**, *"A Large-Scale Analysis of the Security of Embedded Firmwares"*, short paper presentation at SSTIC, Rennes France, June 2015.

2. **A. Costin**, J. Zaddach, *"Analysis of Security of Embedded Devices"*, invited talk at SECURE.PL, Warsaw Poland, October 2014.

3. J. Zaddach, **A. Costin**, *"Embedded Devices Security and Firmware Reverse Engineering"*, tutorial/workshop at Black Hat, Las Vegas USA, August 2013 [204].

4. **A. Costin**, J.Zaddach, *"POSTER: Firmware.RE: Firmware Unpacking and Analysis as a Service"*, poster presentation at the ACM Conference on Security and Privacy in Wireless and Mobile Networks (ACM WiSec), Oxford UK, July 2014.

5. **A. Costin**, *"A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design"*, invited talk at HITB, Kuala-Lumpur Malaysia, October 2014.

6. **A. Costin**, *"A Dangerous 'Pyrotechnic Composition': Fireworks, Embedded Wireless and Insecurity-by-Design"*, invited talk at DefCamp, Bucharest Romania, November 2014.

7. **A. Costin**, A. Francillon, *"Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices"*, invited talk at INSA, Lyon France, March 2013.

8. **A. Costin**, A. Francillon, *"Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices"*, invited talk at Black Hat, Las Vegas USA, August 2012.

9. **A. Costin**, *"Security of Network Monitoring Systems (NMS) for Cloud and HPC. Hands-on assessment, analysis and countermeasures."*, invited lecture at "COINS Summer School 2015 on Cloud Security", Lesbos, Greece, August 2015.

Chapter 1

Introduction

Embedded systems are omnipresent in our everyday life and they 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 [133, 170]. Cisco famously predicted that there will be 50 billion of *connected embedded devices* by 2020 [72]. Those devices will be produced by many different manufacturers and will be present in many different models. Each will probably have several firmware versions, leading to an overall huge number of firmware releases. As we show in Chapter 4, hundred of thousands of firmware images are already available, which is just a lower bound estimate of publicly observable firmware packages. The number of firmware files will likely only grow with the number of new embedded devices being developed and deployed.

At the same time, the security of an average embedded device's firmware is empirically shown to be often weak [115, 198]. This has been frequently shown by independent evaluations [58, 129, 132, 162]. Such evaluations often show that the security of many embedded devices and their firmware is very low. That once again proves that many vendors are usually more interested in fastest and cheapest release of new products and features to increase their market share. This practice is usually opposite to building secure products and accurately testing them against current and future security threats. These facts are even more worrying because the security flaws in the embedded devices and their firmware are many times found by security practitioners using approaches that are neither systematic nor automated [36, 121].

Moreover, vulnerabilities in the firmware constitute an easy entry point for malicious software and make the embedded devices prone to simple yet devastating attacks. In fact, since 2009, multiple botnets have been discovered that exploited various firmware vulnerabilities. Such botnets have compromised thousands, if not millions, of online embedded devices [49, 65–67, 99, 174, 195, 196]. Even worse, the affected embedded devices are hard to diagnose and clean (e.g., no embedded anti-virus solutions, no conventional input/output). Therefore, they often remain

exploited for long periods of time. For example, the `Carna` botnet [65] which was used to produce the (in)famous "Internet Census 2012" was operational for more than one year. In addition to this, the rate of embedded devices expected to connect to the Internet is exponential and the speed at which attacks can spread across systems and networks is unimaginable. For example, the `Slammer` worm infected more than 90% of the vulnerable machines within 10 minutes [158]. As a consequence, manual intervention or analysis is hard, if not impossible. This confirms the need for detecting vulnerabilities in firmware images before they are exploited by attackers. Manual firmware analysis can find such problems [121], however it can be a lot more efficient to automate the process. In this context, it is desirable that the security analysis of firmware packages be automated and fast, and be performed continuously and on a large scale.

The situation is expected to become even more troubling for multiple reasons, which can be explained using a recent analysis of the expected IoT evolution [86]. First, by 2017 the number of connected IoT devices alone is expected to surpass the PC, tablet and phone markets combined, with a global IoT device installed base of around 7.5 billion devices. Second, the IoT devices are predicted to be more or less equally distributed (i.e., by device count) between enterprise, government (e.g., critical infrastructure) and home sectors [86]. This means that all the major sectors are expected to be exposed to security threats arising from vulnerable embedded devices. It is interesting to note that the predicted scenario is somewhat similar to the surge in the mobile attacks and malware in late 2011 [154]. This happened approximately when the number of mobile devices in use (i.e., smartphones, tablets) surpassed the number of PCs [86]. It is very likely that by 2017 the IoT and embedded devices will face similar attacks and security scrutiny the way mobile technologies did back in 2011. However, this can probably happen at a much larger scale and having a high impact.

It is known that present techniques are not completely adequate to effectively and efficiently discover firmware vulnerabilities in a scalable manner. Some techniques, such as `Avatar` [203], often require physical access to devices and laborious manual setup for each device. Other techniques, such as `Firmalice` [184], require a security policy to be provided for each device. Therefore, by using present methods, manual work is almost always necessary and large-scale studies are infeasible.

Additionally, in order to achieve complete automation and a continuous improvement of the analysis process, two more stages must preferably be automated. Firstly, as more firmware packages are released and hence collected, the ability to accurately classify them or tell firmware from non-firmware apart becomes important. For example, this can be useful to automatically cluster firmware images for the same device or vendor, and then analyse them using methods specific to the device or the vendor. Unfortunately, the current efforts to detect and classify files (e.g., malware) [43, 145, 180], or to use machine learning [182, 191], are limited to specific areas and do not cover specifics of embedded devices and

their firmware. Secondly, as more embedded devices are getting web-enabled and connected to the Internet, the ability to fingerprint and accurately identify them becomes important. For example, this can be useful to quickly identify and isolate populations of embedded devices affected by a particular vulnerability. However, current techniques cannot be easily applied [60, 90, 113] to embedded devices that are web-enabled or connected to the Internet. Therefore, even though automating these two stages is not explicitly related to pure security analysis and vulnerability discovery, they are desirable in a large scale automated setup.

All the above considerations are the basis of the increasing need for large scale automated techniques to achieve two main goals. One is to perform effective firmware security analysis. Another is to accurately classify connected embedded devices and collected firmware files.

1.1 Contributions of the Thesis

This dissertation describes scalable techniques to discover vulnerabilities in embedded firmware and to classify firmware packages and live embedded devices. The methodology we propose is based on automated and flexible mechanisms. The initial step is to greedily and continuously collect a large number of heterogeneous firmware packages. Then we developed techniques for efficiently unpacking the firmware and statically analyze the unpacked files. We then try to emulate each firmware and its services (including the embedded web interfaces) in a "best effort" and generic way. Subsequently, we applied dynamic analysis (e.g., web application security tools) on each emulated instance of the firmware. An additional step is the correlation of common vulnerabilities among the population of firmware packages. Finally, we applied Machine Learning (ML) to classify firmware files and web application fingerprinting to classify live devices. The end-to-end methodology is depicted in the Figure 6.6.

Like other vulnerability discovery and classification methodologies, our technique does not guarantee to provide a full coverage of the vulnerability discovery. Nor does it guarantee a completely accurate classification of firmware files and embedded devices. Nevertheless, it is the first demonstration of the feasibility of large scale vulnerability discovery in firmware packages. Therefore we claim it is an effective way to help increase the security of embedded devices, and hence of the IoT. Even though we mainly limit the vulnerability discovery scope to Linux-based firmware images (for ARM, MIPS and MIPSEL architectures), the very same techniques could be easily extended to other CPU architectures (e.g., PowerPC), OSes (e.g., VxWorks) and software distributions. In addition, other analysis methods or tools could be easily integrated with our framework. For example, symbolic execution engines or fuzzing tools could be used to perform advanced dynamic analysis.

We have developed a fully automated framework and used it to test vulnerability discovery at large scale. Our system was able to find statically 38 new vulnerabilities in 693 firmware packages. In addition to this, our system was able to discover dynamically 225 high-impact vulnerabilities (e.g., command injection, XSS) in at least 20% of emulated embedded web interfaces (i.e., 45 firmware packages). We also used the framework to test automated firmware and device classification. Our automated system was able to correctly classify firmware packages and identify live devices with an accuracy of 90% or more.

The contributions within this dissertation can be summarized as follows:

- We are the first to propose and perform the collection and the security analysis of embedded firmware images at large scale.
- We formulated the first five core challenges associated with this type of research, namely: Building a Representative Firmware Dataset; Firmware Identification; Firmware Unpacking and Custom Formats; Scalability and Computational Limits; Results Confirmation.
- We are also the first to demonstrate the feasibility to fully automate dynamic analysis of heterogeneous embedded firmware at scale. We demonstrate this with large scale dynamic analysis of embedded web interfaces.
- Moreover, we propose the first large scale and highly accurate classification of firmware packages and live embedded devices. For this, we apply Machine Learning (ML), web interface level fingerprints and multi-metric score fusion.
- We implement the proposed methods in a fully automated framework that allowed us to quickly find in practice a large number of new vulnerabilities in many firmware packages for a variety of device classes and vendors.
- Finally, we propose firmware collection, unpacking and analysis as a service (`http://firmware.re`).

1.2 Organization

The rest of the dissertation is organized as follows:

- In Chapter 2 we survey the state of the art relevant to our work, in which we provide a description of the existing publications, tools and experiments presented by both academia and industry.
- In Chapter 3 we present an end-to-end case study of the (in)security analysis of wireless pyrotechnic systems. We use this case study as a motivating example for our main work in the following chapters.

- In Chapter 4 we introduce our methodology and describe the large scale analysis framework that we implemented. We also provide early insights on results and challenges.
- In Chapter 5, based on the established framework, we focus on the analysis of the embedded web interfaces using firmware emulation combined with static and dynamic techniques. We show how our framework can be used in practice to quickly find new vulnerabilities in embedded web interfaces.
- In Chapter 6 we further focus on the automated and accurate classification of both firmware files and online embedded devices. We present our experience with exploring possible feature sets and fingerprint metrics. We also discuss applying Machine Learning (ML) and multi-metric score fusion techniques for automated and accurate classification at large scale.
- Finally, Chapter 7 concludes the dissertation and presents possible further improvements of our work.

Chapter 2

State of The Art

During the last few years the problem of systematically securing the embedded devices and their firmware generated an increasing interest from both academia and industry.

This problem is complex due to several factors. For example, different embedded devices and firmware images have different security requirements and protections, operating environments, hardware architectures and software support. Additionally, they may have different security expectations, and might implement (or not) the security requirements in a variety of ways.

For these reasons, we present in this chapter a survey of research directions, methodologies and tools that have been designed to perform security analyses of embedded devices and firmware images from an interdisciplinary perspective.

2.1 Real-World Studies

2.1.1 Large Scale Studies of Embedded Devices and Firmware Images (In-the-Wild Approach)

Several studies have been proposed to asses the security of embedded devices by scanning the Internet. Cui et al. [83] presented a wide-scale Internet scan to recognize devices that are known to be shipped with default credentials, and subsequently to confirm that the discovered devices are indeed still vulnerable by attempting to login into them. Similarly, the (in)famous *Internet Census 2012* [26] is an anonymous research project that took advantage of insecure embedded devices to build a large scale distributed botnet [65]. The authors (ab)used the devices in the botnet to perform an entire IPv4 scan using stack fingerprinting based on (or similar to) NMAP [114]. Unfortunately, it is not completely clear how ethical and legal these Internet surveys were. Heninger et

al. [124] performed the largest ever network survey of TLS/SSL and SSH servers. Their survey showed that vulnerable or weak keys are surprisingly widespread and that the vast majority appear to belong to headless or embedded devices. ZMap [102] is an efficient and fast network scanner, that allows to scan the complete Internet IPv4 address space in less than one hour. Even though the scans are not especially targeting embedded devices, many such devices are present in this dataset. In Chapter 4 we reuse the HTTPS/SSL certificates scans that were performed using ZMap [101].

Some online services like Shodan [155] perform regular Internet scans and provide a global 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. At the same time, many existing projects scanned the Internet, or parts of it, to discover vulnerabilities in embedded systems [26, 83, 124, 155, 159, 159]. Such wide scale scans are mainly aiming to discover online devices affected by already known vulnerabilities. Although, in some cases they can help discover new flaws [124], many categories of vulnerabilities cannot be in principle discovered by such scans.

Zheng et al. [207] performed the first large scale analysis of customized Android firmware images. They used both static and dynamic analyses to evaluate the firmware security on both the application and system levels The authors collected 250 customized Android firmware images containing around 24K pre-installed applications. They also investigated a real-world large scale attack on around 348K Android devices involving a pre-installed zero-day malware known as `CEPlugnew`.

2.1.2 Small Scale Studies of Supervised Embedded Devices (In-the-Lab Approach)

Large scale and in-the-wild studies have many benefits and can provide unique insights, as outlined above. However, one main limitation is the lack of complete control over the embedded devices or the experimental environment. On the contrary, small scale and in-the-lab controlled environments are much easier to manage and observe. They allow in-depth analysis and thus are able to provide more detailed results.

To date, several reports exist that document in-the-lab small scale studies performed on live embedded devices. First, Bojinov et al. [58] conducted an assessment of the security of current embedded management interfaces. The study was conducted on real physical devices. The authors found vulnerabilities in 21 devices from 16 different brands, including network switches, cameras, photo frames, and Lights-Out Management (LOM) modules. In a similar study, a security lab manually analyzed ten Small-Office Home-Office (SOHO) routers [132] and they discovered at least two vulnerabilities per device. Their research revealed

in total 55 previously undisclosed security vulnerabilities in all ten devices. The authors experimentally demonstrated how flaws in services that are not essential can lead to full compromise of the routers and discussed possible mitigations for these vulnerabilities. Recently, in another study [129] ten of the most popular devices such as TVs, door locks and home alarms, were manually reviewed for security issues. The study revealed an alarmingly high average number of vulnerabilities per device. Vulnerabilities ranged from weak passwords and Cross-Site Scripting (XSS) to `Heartbleed` [100] and Denial of Service (DoS). Finally, Niemietz and Schwenk [162] studied the security of embedded web interfaces of ten popular DSL home routers. Authors found that all ten are vulnerable to UI redressing and eight suffer from XSS vulnerabilities.

Indeed, the sample size of all these studies is quite small to be representative or statistically significant. Even so, they confirm the general perception that many embedded devices often contain important vulnerabilities.

2.2 Firmware Analysis

2.2.1 Firmware Unpacking, (Re)Packing and Malicious Modifications

To perform a successful analysis of a firmware, it requires locating and extracting important functional blocks (e.g., binary code, configuration files, scripts, web interfaces) from the firmware package. 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. However, embedded software distribution lacks general standards and vendors have often 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. Many firmware images are based on *file system images* where the bootloader, the operating system kernel and the applications are well structured and separated. These are frequently Linux-based firmware images which are, in general, easy to unpack. However, there are also *monolithic firmware images* where the bootloader, the operating system kernel, the applications, and other resources are combined together in a single memory image. These are especially challenging to unpack.

Therefore, unpacking firmware images is a known problem and several tools for this purpose exist. `Binwalk` [117] 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` [81] is a framework to unpack, analyze, and repack firmware images of embedded devices. `FRAK` was never publicly released and reportedly supports only very few firmware formats (e.g.,

Cisco IOS and IP phones, HP LaserJet printers). The Binary Analysis Toolkit (`BAT`) [123, 192] 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.

Looking at insecure (remote) firmware updates, researchers reported the possibility to arbitrarily inject malware into the firmware of a variety of devices. Stamm et al. [186] demonstrated that malicious modifications of a D-Link DI-524 router's firmware can be used to mount powerful attacks. For example, attackers could executed drive-by pharming attacks [186] in which they take advantage of an inadequately protected router to gain access to user data or to change router's settings (e.g., DNS). Such attacks could result in Denial of Service (DoS), malware infection or other unwanted results. Authors also suggested that router-based botnets (see Section 2.2.2) could be built using these techniques. Several other researchers presented techniques and implications of exploiting Apple firmware updates. Chen [70] reversed Apple keyboard firmware updates and demonstrated how to achieve a persistent rootkit. Miller [157] reverse engineered the firmware and its "flashing" process for a particular Apple smart battery controller. The author showed how to reprogram the smart battery by modifying its firmware, and argued this may be enough to cause safety issues such as producing fire hazards. Brocker and Checkoway [63] demonstrated it is possible to disable the "camera ON" physical LED indicator via malicious firmware modifications of the iSight cameras. In other experiments, several network card firmware images have been analyzed and modified to insert a backdoor [89, 97] or to extend their functionality [54]. Basnight et al. [46] examined the vulnerability of PLCs to intentional firmware modifications. The authors presented a general firmware analysis methodology, and experimentally demonstrated how legitimate firmware can be updated on an Allen-Bradley ControlLogix L61 PLC. Costin [73, 74] first demonstrated the firmware modification attacks on printers. To accomplish this the author (ab)used flaws in multiple protocols such as HP Remote Firmware Update (RFU), Printer Job Language (PJL) and PostScript. Delivering the attack via standard printed documents, Costin was able to have full access to the underlying `VxWorks` OS and demonstrated among other things the ability of outbound communication to a malicious server. Cui et al. [82] discussed how the firmware update feature could be (ab)used by the attackers to inject malicious firmware modifications into vulnerable embedded devices. Using the techniques from [73, 74], the authors presented a case study of a firmware modification vulnerability in the HP RFU protocol to arbitrarily inject malware into HP LaserJet printers. Zaddach et al. [205] explored the consequences of a backdoor injection into the firmware of a hard disk drive and its use to exfiltrate data.

2.2.2 Malware for Embedded Devices and Firmware Images

Recently, the botnets and the worms started to actively target the embedded devices. This is not surprising given the security state of the embedded devices, such as the COTS embedded devices, is commonly bad. Therefore, botnets and worms exploit known or 0-day vulnerabilities in the firmware of the affected devices. For example, they often exploit default and hardcoded credentials, or unauthenticated remote command and code execution. This worrying trend motivates an overview of existing botnets exploiting firmware vulnerabilities in embedded devices.

In 2009, the `psyb0t` botnet was discovered [49, 99]. It was operating a new form of malware which was specifically targeting MIPS Linux based ADSL routers. It affected thousands of insecure devices and the botnet was used primarily as a proof-of-concept as well as to launch several DDoS attacks. Embarrassingly, the `psyb0t` threat could have been mitigated through a firmware update and a change in the default authentication credentials. The `Chuck Norris` botnet [67] exploited poorly-configured or outdated firmware MIPSEL Linux devices, in particular ADSL modems and routers. The botnet attacked devices that exposed the `telnet` service. It gained `telnet` access by brute-forcing credentials and then used this access as an infection vector to extend itself. The Linux malware operated by this botnet was controlled by a central Command-and-Control (C&C) server, while the infected devices were coordinated over IRC channels. Andersson and Szewczyk [40] presented a detailed overview of malware risks associated with insecure or improperly administered ADSL routers. The authors very well summarized many studies referring to the `psyb0t` and `Chuck Norris` botnets operating over infected embedded devices.

The `Aidra` (a.k.a `Hydra`) botnet [66] started to emerge in 2012. It is built using an open source IRC-based mass router that contains scanning and exploitation modules. Its source was made publicly available for download from the Internet [107]. The novelty of this botnet is that it supports six CPU architectures (x86, ARM, MIPS/MIPSEL, PPC, SH4).

The `Carna` botnet [65] consisted of around 400K embedded devices "infected" with benign software. It targeted multiple CPU architectures and device classes. Similarly to `psyb0t`, it abused the default credentials in many connected embedded devices to propagate itself. In one instance, the `Carna` botnet was used by the anonymous researcher who created it to produce the (in)famous "Internet Census 2012" [65] survey dataset.

`TheMoon` worm [174, 195, 196] emerged in 2014. It targets MIPS based devices and currently affects Linksys (Belkin) E-series routers. It spreads by sending an exploit to a vulnerable CGI script running on these routers. The worm exploits an unauthenticated remote command injection flaw in an administrative script that fails to properly check the credentials.

Bitcoin mining is the latest trend in embedded malware, exploitation and botnet operation. In 2014, Symantec documented `Linux.Darlloz`, an IoT worm used to mine crypto currencies [189]. It affected around 31K devices and targeted routers, Set-Top Boxes (STB), IP-cameras running ARM, MIPS and PowerPC architectures. Similarly, in 2014 SANS Technology Institute uncovered and published malware samples targeting ARM-based Digital Video Recorders (DVR) [197] and MIPS-based routers [194] for the purpose of mining digital currencies.

Finally, Minn Pa Pa et al. [167] analyzed the increasing threats against embedded devices. The authors showed that starting with 2014, Telnet-based attacks that target IoT devices have rocketed. They also proposed and setup `IoTPOT`, a honeypot and sandbox to attract and analyze Telnet-based attacks against various IoT devices. The authors observed in total four IoT malware families, some of them supporting up to nine CPU architectures.

2.2.3 Static and Dynamic Firmware Analysis

To date there are several separate examples of security analysis of embedded systems.

Performing such studies requires static and dynamic techniques and tools specifically developed for embedded devices and their firmware images. Davidson et al. [87] proposed `FIE`, a firmware analysis tool built on top of `KLEE` symbolic execution engine [64]. It incorporates new symbolic execution techniques and can be used to verify security properties of some simple firmware images often found in practice. Zaddach et al. [203] described `Avatar`, a dynamic analysis platform for firmware security testing. In `Avatar`, the instructions are executed in an emulator, while the input and output accesses to the peripherals of the embedded system 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. Shoshitaishvili et al. [184] presented `Firmalice`, a static binary analysis framework to support the analysis of firmware files for embedded devices. Authors developed a model to describe, in an architecture- and implementation-independent way, authentication bypass vulnerabilities in firmware binaries. They also implemented a tool that uses advanced program analysis techniques to analyze binary code in complex firmware of diverse hardware platforms. Authors verified their technique on three *known backdoors* in real devices. Binary static analysis can be successfully applied to binary CGIs to find vulnerabilities such as buffer overflows, (remote) code executions, command injections (e.g., `Firmalice` [184] or `WEASEL` [181]). Finally, new techniques start to appear that are able to cope with the diversity of CPU architectures found in embedded systems. For instance, Pewny et. al [169] proposed a system to derive bug signatures for known vulnerabilities. They use an intermediate representation of the buggy code and can support x86, ARM and MIPS architectures. The authors showed their system can find the `Heartbleed` vulnerabilities regardless of

the underlying software instruction set. They also applied their method to find backdoors in firmware images of routers running ARM or MIPS.

2.2.4 Firmware Emulation

There are several recent approaches that rely on emulation in order to discover or verify vulnerabilities in embedded systems. Davidson et al. [87] present `FIE`, which intended to verify security properties of some simple firmware images often found in real-world as well as to discover vulnerabilities in such firmwares. It is based on the `KLEE` symbolic execution engine and allows easier incorporation of new symbolic execution techniques. `FIE` emulates the firmware under analysis by translating it into LLVM bitcode and then emulating it.

Kammerstetter et al. [141] developed `Prospect`. It targets Linux-based embedded systems that are emulated with a custom kernel which forwards `ioctl` requests to the embedded device that runs the device's normal kernel. Li et al. [151] proposed `FEMU`, a hybrid firmware/hardware emulation framework to achieve confident System-on-Chip (SoC) verification. The authors used a transplanted QEMU at BIOS level to directly emulate devices upon hardware. While not oriented towards security aspects of SoC, on a practical SoC project it helped authors identify several design issues in full-system emulation. Dolan-Gavitt et al. [91] presented `PANDA`, a tool based on QEMU emulator that was built to support whole system emulation and reverse engineering. Koscher et al. [149] presented `Surrogates`, a system that can emulate and instrument embedded systems in near-real-time. It also enables a variety of dynamic analysis techniques. `Surrogates` framework provides the emulator with an accurate representation of the environment where the firmware is being executed. Finally, `Avatar` also provides firmware emulation for security analysis. It executes embedded code inside a QEMU emulator, while the input and output requests to the peripherals of the embedded system are forwarded to the real device attached to the framework.

2.3 Web-related Aspects

Many embedded devices are designed to be inherently connected. Moreover, the IoT and the Web-of-Things (WoT) paradigms assume that the embedded devices are connected and are web-enabled. It also almost always implies the presence of a web-interface. In this section we therefore discuss the security of web interfaces inside embedded devices and their firmware.

2.3.1 Web Application Security

Web application security is a well established research field with extensive prior work. Therefore, we focus on aspects of this field which directly relate to the

work presented in this thesis.

Huang et al. [131] were the first to statically search for web vulnerabilities in the context of PHP applications. They used a lattice-based analysis algorithm derived from type systems and type state and found many SQL injection and XSS vulnerabilities in PHP code. Pixy [139] proposed a technique based on data flow analysis for detecting XSS, SQL or command injections. RIPS [85], on the other hand, is a static code analysis approach that uses tainting to detect multiple types of injection vulnerabilities. Bojinov et al. [58] studied the security of embedded management interfaces and a new class of vulnerabilities was discovered, namely Cross-Channel Scripting (XCS) [57]. Bencsáth et al. [52] demonstrate that a full compromise of embedded devices is indeed possible in practice when XSS infected pages are opened by the device administrator. The XSS exploitation in turn uses the XCS vulnerabilities to accomplish this attack on real embedded devices. Balzarotti et al. [44] showed that even if the developer performs certain sanitization on input data, often XSS attacks are still possible due to the deficiencies in the sanitization routines. In their work they checked whether sanitization routines are sufficient, not just that they are present. For this, they described `Saner`, a combined static and dynamic analysis to find such security bugs.

Fong and Okun [112] took a closer look at web application scanners, and their functions and definitions. The authors proposed a taxonomy of software security tools and described the types of functions that are generally found in a web application scanner. Bau et al. [48] conducted an evaluation of the state of the art tools for automated "black box" web application vulnerability testing. While results have shown the promise and effectiveness of such tools, they also proved many limitations of existing tools. For example, the authors found that variants of XSS and SQLi vulnerabilities were missed by many tools. Similarly, Doupé et al. [95] performed an evaluation of eleven "black box" web application vulnerability testing tools, both open-source and commercial. The authors found that crawling ability is as important and challenging as vulnerability detection techniques and many classes of vulnerabilities are completely overlooked. They concluded that more research is required to improve the tools and the techniques behind those tools. Holm et al. [127] performed a quantitative evaluation of vulnerability scanning. The authors showed that automated scanning is unable to accurately identify all vulnerabilities. They also show that scans of Linux-based hosts are less accurate than the scans of Windows-based ones. Doupé et al. [94] proposed improvements to the "black box" vulnerability testing tools. First, they observed the web application state *from the outside*, which allowed them to extend their testing coverage. Then they closely controlled the "black box" web application vulnerability scanner. They implemented their technique in a crawler linked to a fuzzing component of an open-source web application vulnerability testing tool.

Gourdin et al. [116] addressed the challenge of building secure embedded web in-

terfaces by proposing `WebDroid`, the first framework specifically dedicated to this purpose. Authors experimentally demonstrated the efficiency of their framework in terms of performance and security.

2.3.2 Web Application Fingerprinting and Identification

Web application fingerprinting is also a well explored research field. Currently, several techniques and tools are available for fingerprinting the web applications.

Shah [183] presented early techniques to fingerprint and identify web applications at the HTTP level. The author proposed a theoretical fingerprint logic based on multiple methods such as Decision Trees (DT), statistical analysis and Neural Networks (NN). The author also implemented some of the techniques into the `httprint`[1] tool. Unfortunately, no evaluation of the accuracy was performed, moreover, the `httprint` tool and signatures have not been updated since 2005. Similarly, the BlindElephant [55] tool attempts to discover the version of a web application by comparing static files at known locations against precomputed hashes of those files in all known available version releases. Wapplyzer [200] is a browser plugin which uses regular expressions to uncover the technologies used on websites and within web applications. On a similar line, WhatWeb [160] uses more than 900 plugins to recognize the web technologies used within a website.

On the other hand, Alvarez [39] used the Extended File Information (EXIF) metadata in JPEG files to generate fingerprints. Likewise, Bongard [59] studied the implementation differences among the PNG codecs used with the most popular web application development platforms. The authors proposed to use these differences to fingerprint and classify web applications.

Salusky and Thomas [176,177] disclosed processes for fingerprinting and identifying client applications based on the analysis of their HTTP requests. The authors constructed the fingerprint based on the presence and the order of HTTP headers included in a request from a client application or device. Authors showed that this can be used to assess if a client application is malicious. Similarly, Zarras et al. [206] monitored the HTTP requests from web clients on network level and were able to fingerprint the clients by detecting subtle differences in the implementation of the HTTP protocol of each client.

2.4 Fingerprinting and Classification

Fingerprinting and identification is an established research field with supported by extensive research. Below we discuss existing work related to this field in the context of embedded devices and their firmware files.

[1] http://www.net-square.com/httprint.html

2.4.1 Embedded Device Fingerprinting and Identification

Kohno et al. [144] presented a technique for remote physical device fingerprinting by exploiting the fact that modern computer chips have small but remotely detectable clock skews. Rasmussen and Capkun [60] demonstrated the feasibility of radio fingerprinting of wireless sensor nodes. Their detection scheme extracts the radio signal transient and its features. The authors were able to create radio fingerprints and then identify the origins of the messages. Desmond et al. [90] proposed a wireless fingerprinting technique that differentiates between unique devices through timing analysis of 802.11 probe request frames. Franklin et al. [113] developed a passive fingerprinting technique that identifies the wireless device driver running on an 802.11 compliant device. The authors exploited the fact that most wireless drivers have implemented different active scanning algorithms.

On the other hand, Eckersley [103] conducted a study showing that various properties of a user's browser and plugins can be combined to form a unique fingerprint of the browser. Subsequently, Nikiforakis et al. [163] examined how the heavy use of Adobe Flash can lead to web-based device fingerprinting on the Internet.

Recently, Niemietz and Schwenk [162] used manual fingerprinting of the web interfaces of ten popular DSL home routers. The authors employed techniques such as existence of unique or particular files and URLs, as well as presence of distinguishable strings in *HTTP Basic Authentication* and *Web Interface Authentication* prompts. Cui and Stolfo [83] performed a wide-area IPv4 scan and presented a quantitative analysis of the insecurity of embedded devices. While the authors found around 540K embedded device connected to the Internet, they could attribute them to as few as 73 device types which mapped to only 9 functional categories (e.g., VoIP Devices, Home Networking Devices, Camera/-Surveillance). Unfortunately, the authors did not detail their fingerprinting and classification methods that lead to their categorization. Also, the small number of device types (i.e., 73) and functional categories (i.e., 9) compared to the large number of devices (540K), suggests that the authors likely used coarse-grained and approximate methods to fingerprint and classify the devices.

2.4.2 File Classification

Fuzzy hashing aims at comparing two different objects (e.g., files, forensic memory dumps) and provides a measure of their similarity. The two most popular fuzzy hashes are `sdhash` [175] and `ssdeep` [146]. French and Casey [47] presented fuzzy hashing techniques in applied malware analysis and classification. The authors used `ssdeep` on the *CERT Artifact Catalog* database containing $10.7M$ files. The study underlined the two fundamental challenges in operational usage of fuzzy hashing at scale.

Bailey et al. [43] and Bayer et al. [50] proposed efficient clustering approaches to identify and group malware samples at large scale. The authors performed 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 currently applied in our framework since dynamic analysis is unfeasible due to the heterogeneity of architectures used in firmware images.

2.5 Summary

In this chapter, we presented the related work relevant to the security of embedded devices and their firmware. We first introduced the main real-world security studies of embedded devices and their firmware. Some of these studies were performed at a large scale. However, these were not particularly aimed at firmware analysis but rather at embedded devices discovery and limited firmware testing.

We then presented the main efforts in firmware analysis. Recently, a number of papers on dynamic firmware analysis and firmware emulation emerged. Those papers bring useful insights, but often require manual intervention and present many challenges to automate completely. In this section we also showed that malware developers are also performing firmware analysis. Then they employ known or unknown firmware vulnerabilities to build malware for and botnets of vulnerable embedded devices.

Then we reviewed the recent work in web application security. We presented the main efforts and limitations of the web application security field. We also presented existing techniques for web application fingerprinting. Subsequently we reviewed the web application security in the context of embedded web interfaces.

We concluded the related work section by discussing recent work on (remote) device fingerprinting, and automated file classification and labeling.

To the best of our knowledge, none of these efforts have looked at scalable techniques to study the firmware for embedded devices. Also, no empirical studies have been previously conducted to define firmware unique identifiable properties and representative datasets, prevalence of vulnerability categories, challenges encountered, and open problems.

The analysis techniques introduced in this dissertation attempt to overcome these limitations. We achieve this by developing methods that scale in terms of dynamic and static analysis, and also in terms of machine learning and classification. With our techniques, we try to address some important challenges and missing points in large scale analysis of embedded devices. In particular, in the following chapters

we try to address: the firmware and embedded devices identification in big-data scenarios; the scalability of the dynamic and static firmware analysis; and finally the security of the embedded web interfaces.

Chapter 3

Motivating Example – Insecurity of Wireless Embedded Pyrotechnic Systems

3.1 Introduction

Fireworks are essentially explosives used for entertainment purposes. A *fireworks event*, also called a *pyrotechnic show* or *fireworks show*, is a display of the effects produced by *fireworks devices*. Fireworks devices are designed to produce effects such as noise, light, smoke, floating materials (e.g., confetti). The fireworks event and fireworks devices are controlled by *fireworks firing systems*. Firing systems, besides fireworks, are often used for other purposes, such as building demolition, special effects, and military training or simulation.

Despite the fact that fireworks are intended for celebrations, their usage is often associated with high risks of destruction, injuries, and even death. Many recent news and research studies show the dangers of fireworks [17, 172]. Sometimes fireworks are even used as real weapons in street clashes [33]. Fireworks accidents are often caused by equipment mishandling, not following safety rules or low quality of the fireworks devices. Another aggravating factor is that fireworks are generally intended to be displayed in densely crowded and public areas. All these accidents still happen despite the strict control of the distribution of fireworks and the need for a professional license to handle such devices.

Classically *fireworks firing systems* consist of mechanical or electrical switches and electric wiring (often called shooting wire). This type of setup is simple, efficient and relatively safe [24]. However, it dramatically limits the effects, complexity and capabilities of the fireworks systems and events. Advances in software,

embedded and wireless technologies allows fireworks systems to take full benefit of them. A modern (wireless) firing system can be considered to be a good example of a complete *Embedded Cyber-Physical System (ECPS)* or *Wireless Sensor/Actuator Network (WSAN)*. Fireworks firing systems are increasingly relying on wireless, embedded and software technologies. Therefore, they are exposed to the very same risks as any other ECPS, WSAN or computer system.

Based on recent research, both critical and embedded systems of all types acquired a bad security reputation. For example, airplanes can be spoofed on new radar systems [76], a car control can be taken over [69, 148] and can be compromised to failure [126], an implanted insulin pump can be completely compromised [173] or an array of PLCs in a nuclear facility can be rendered nonfunctional [109, 150].

In this chapter we approach the study of firing system risks from the perspective of computer, embedded and wireless security. We describe our experience in discovering and exploiting a wireless firing system in a short amount of time without any prior knowledge of such systems. In summary, we demonstrate our methodology starting from analysis of firmware to the discovery of vulnerabilities. Our static analysis helped our decision to acquire such a system which we analyzed in-depth. This allowed us to confirm the presence of exploitable vulnerabilities on the actual hardware. Finally, we stress on the need of hardware and software security and safety compliance enforcement for pyrotechnic firing systems.

3.2 Fireworks Systems Architecture

Figure 3.1 presents a generic diagram of a fireworks firing system. A fireworks firing system is composed of:

- *Remote control modules* (also sometimes known as *main control*) – these control the entire show, which includes sequencing cues and sending *fire* commands. They connect to firing modules by wired or wireless connections. In simple scenarios a single remote control module is paired with all firing modules, while in more complex shows there are several remote control modules, each one paired with a show-specific subset of firing modules. All remote control modules act independent of each other. Those devices rely on a microcontroller embedding its own firmware.
- *Firing modules* – these receive *fire* commands from remote control modules and activate minimum ignition current for the igniter clips. Firing modules are based on micro-controllers and also have their own firmware.
- *Wired connections* – these are described here for completeness. However, they do not apply to our case study where remote control and firing modules are all wireless. Classic *fireworks firing systems* consist of electric

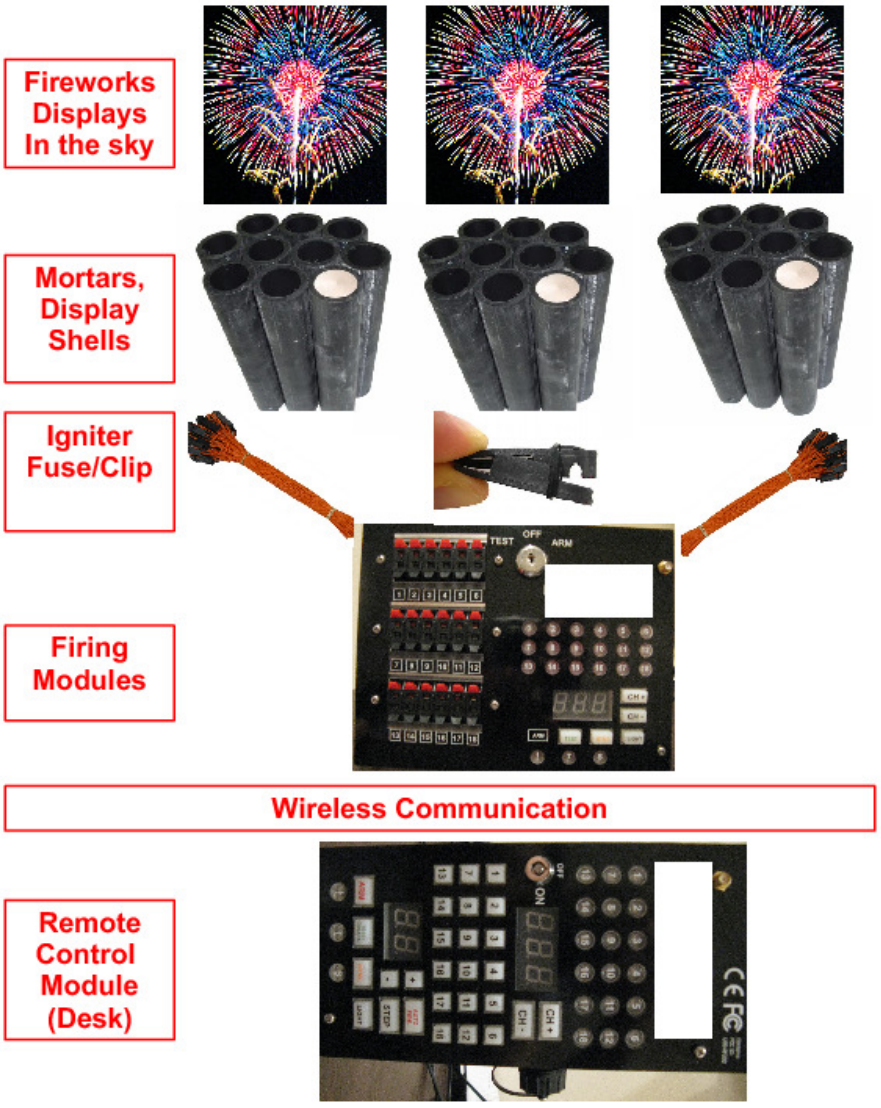

Figure 3.1: Generic diagram and components of a wireless firing system.

wiring between remote control and firing modules [24]. Simple connection cables having End-Of-Line (EOL) resistors are used to securely terminate wire loops. EOL resistors allow the remote control to monitor the field wiring for open or short circuit conditions, hence detecting wiring problems and tampering.

- *Wireless transceivers* – these enable the wireless connections between the remote control modules and the firing modules. Those connections are often performed using 433.92 MHz modules (often capable of using *rolling codes* [15]), or 2.4GHz ZigBee compatible (IEEE 802.15.4) modules which support AES by standard. Those modules rely on microcontrollers that have their own firmware. The devices we study in section 3.3 are only communicating with *wireless* transceivers between the remote control modules and the firing modules, those actually support AES and several modes of operation, but do not use it.

- *Igniter clips* – these connect firing modules to the pyrotechnic devices housed inside mortars and ignite the fire once firing module activate the minimum necessary current.

- *Mortars* – these house the pyrotechnic devices; they also ensure safe launch and firing of the pyrotechnic device into the sky.

- *Pyrotechnic devices* – these are the actual pyrotechnic compositions which produce visual and sound effects in the sky once *fire* command is activated.

3.2.1 Regulation, Compliance and Certification

Many critical systems, including wireless firing systems, advertise as *"Simple, Reliable, Wireless"* or *"Proven, Secure, Reliable"*. However, such systems must first address regulation, compliance and certifications in order to be able and operate in certain geographical regions.

On the one hand, devices with fire-hazard risk, such as pyrotechnics and explosives, must conform to fire protection regulations of the country of manufacturing and/or operation. For USA, it is the National Fire Protection Association (NFPA). Specifically, NFPA-79 *"provides safeguards for industrial machinery to protect operators, equipment, facilities, and work-in-progress from fire and electrical hazards"* [30]. This standard applies to *"the electrical/electronic equipment, apparatus, or systems of industrial machines operating from a nominal voltage of 600 volts or less"*. The safety feature provided by this standard is the requirement of a key-switched operation before any potentially dangerous action can start.

This certification however does not apply to the hardware designs or the firmware implementations which control NFPA-certified *industrial machinery*.

On the other hand, all wireless or radio-frequency (RF) modules must comply with national radio-frequency licensing and allocation plans. This includes Federal Communications Commission (FCC), CE Marking (*Conformité Européenne*) and Industry Canada (IC) certification. The system we analyze contains a California Eastern Labs (CEL) IEEE 802.15.4 2.4GHz RF transceiver, which is CE and FCC certified. However, those certifications do not apply to the security of the communication channels or network protocols or of the firmware, but only to the transceiver and its frequencies.

We argue that, given the risk of the devices controlled by such equipment, a certification, based on a security evaluation of the architecture, firmware and communications should be mandatory. We show in this chapter that it is not the case.

As a counter-example we consider the avionics field. Avionics encompass virtually the entire spectrum of hardware and software involved in the aviation field where safety and high-risk are considered. All avionics devices must pass strict compliance testing for both hardware (DO-254) and software (DO-178B) [125]. Despite those certifications there are recent examples of wireless avionics protocols shown to be deployed without sufficient security [76].

3.3 Experiments and Results

3.3.1 Summary

As we detail later in Chapter 4, we performed a large-scale firmware analysis by crawling the Internet for firmware images. After unpacking firmware images, we run simple static analysis, correlation and reporting tools on each firmware image. In this process we discovered the firmware images of a wireless firing system [1].

Analysis of firmware images for that system has shown us components (strings, binary code, configurations) which appeared insecure [2]. The findings were convincing enough that we acquired the devices for a detailed analysis. Another factor to motivate the acquisition is that according to the vendor, this system is used by *"over 1000 customers in over 60 countries"*.Hence, these systems appear to be particularly popular among fireworks display companies and can be exploited on large geographical areas and can impact a wide range of public events.

[1] We deliberately omit the name of the vendor and the system for safety and ethical reasons.

[2] This analysis was performed on the stable firmware as of Nov 2013, meanwhile a new firmware addressing most of the security issues we discovered was made stable, and is now deployed.

3.3.2 Firmware Acquisition and Static Analysis

Among many others, our crawlers collected from the Internet several firmware images, in *Intel Hexadecimal Object Files (iHex)* format, dedicated to the the wireless firing system. After unpacking, we use several heuristics, including *keyword matching*. *keyword matching* searches for special keywords such as `backdoor, telnet, UART, shell` which often allows to find multiple vulnerabilities. The firmware images were matching the string `Shell>`. Based on this we isolated those firmware images and proceeded to analyze them further with automated and manual approaches.

We identified several security issues with the firmware images we analyzed. First, plain iHex format does not provide any encryption or authentication. Therefore, the firmware updates are openly accessible for study by the attacker, and likely open to malicious firmware modifications. In addition to this, iHex format provides mechanisms that can be use by attackers to insert code or data into memory regions that might have not been designed to be accessible.

3.3.3 Hardware Acquisition and Analysis

The static analysis findings were convincing enough that we acquired the actual wireless firing system to analyze it further. Indeed, static analysis is known to be faster and to scale better than dynamic analysis as it does not require access to the physical devices. However, one important research challenge remains to confirm the results of static analysis. The analyzed firmware images were designed to run on specific embedded devices, without the actual hardware, it is very hard to confirm the discovered vulnerabilities. Indeed, findings of the static analysis study may be not exploitable in a live system, e.g., because the vulnerable code is not executed, or is activated by a configuration option. Even though this could be discovered by emulation, it is a tedious process in itself as it can be error prone and a generic emulator would need to be customized to emulate this particular platform.

These systems usually come bundled with firing modules and remote control modules. The exact number and placement of each module depends on the setup and choreography of each fireworks show. Both of these modules contain *CEL MeshConnect* 2.4GHz ZigBee (IEEE 802.15.4) transceivers [31]. Both modules are equipped with key-operated switches, as required by NFPA-79 chapter 9.2.

The modules contain chipsets running SNAP which is a Network Operating System (NOS) from Synapse [108]. Also, the modules provide a servicing serial port (UART) which provides access to a built-in menu which displays the `Shell>` prompt we discovered earlier. This allows testing, repairing and debugging the remote control module. The UART ports are only accessible by physically removing the plastic chassis of the modules, thus it can be abused only with physical

attacks. This port could be used for example to manage the AES-128 encryption keys of the wireless ZigBee transceivers. In addition to the above, the USB *SNAP Stick SS200* [34] provides reprogramming and sniffing functions over 2.4GHz ZigBee (IEEE 802.15.4), and is tailored in particular for SNAP chipsets and software.

Remote Control Module

A detailed view of main components of the remote control module can be seen on Figure 3.2. After remote control module's disassembly, we confirmed that it uses a *ColdFire MCF52254* processor from Freescale [29]. This is consistent with the result of *Motorola m68k family* provided by our architecture detection tool in Section 6.2. It also uses a *SST25VF032B* flash chip by Microchip [28].

The remote control module exposes a USB port. This port has two main functions. One function is to upload a fireworks show orchestration script. This orchestrator script is a CSV file which instructs the main processor of the remote control module to which firing module and when to send firing cue signals in order to achieve the planned visual, sound or smoke effects. Another function is to upgrade the firmware of the main (not wireless) micro-controller unit (MCU) of the device. This is done via an `.ihex` file, as described in Section 6.2.

3.3.4 Wireless Analysis

This systems, as many others from other vendors, contains a 2.4GHz ZigBee (IEEE 802.15.4) CEL MeshConnect transceiver. The discovery, configuration query and setup, pairing and firmware upgrade of these units is done through *Synapse Portal* [3] software. We installed *Synapse Portal* and then ran the discovery and configuration query. The wireless chipsets on remote control, firing and firmware reprogramming modules have AES-128 capable firmware installed. However, the encryption is not enabled, no encryption key is present and AES-128 seems to be unused. In addition to this, the system's documentation does not provide any way to configure the system to enable encryption or authentication of the communication. Surprisingly, even though those devices are standard compliant and as such have AES-128 capabilities, neither authentication nor encryption of the messages are used. This is most likely due to the difficulty to properly setup key management and distribution. Such difficulties could be perceived more as a risk of operational failure during a fireworks show, rather than a useful security mechanism.

Further analysis revealed that it is possible to upload Python application scripts to remote wireless chipsets. These scripts are executed in a Python interpreter

[3] http://www.synapse-wireless.com/snap-components-free-developers-IDE-tools/portal

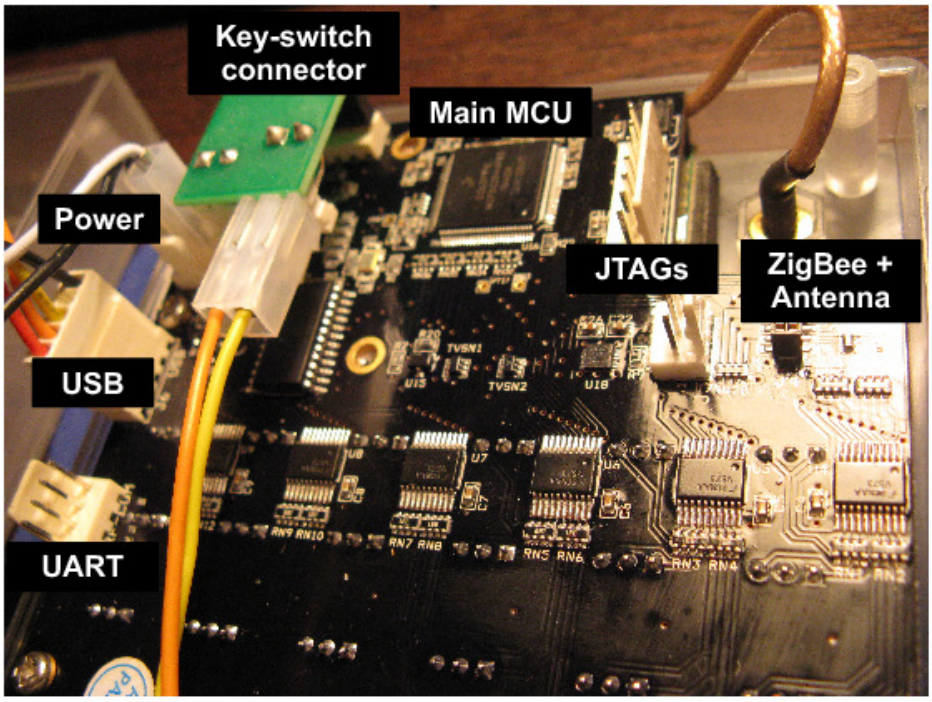

Figure 3.2: Remote control module's hardware.

within the wireless chipset's MCU [31]. The provided interpreter framework is a subset of Python. Before being uploaded to target nodes, *Synapse Portal* compiles these Python scripts into binary form and stores them as SNAPpy files (with extension `.spy`) files [32]. The binary form is targeted for the specific MCU which drives each wireless chipsets. These scripts expose entry-points (functions) that can be remotely called (via RPC) by other wireless nodes. These scripts can interact with the MCU of the wireless chipsets or with GPIO-ports of the wireless chipsets. Usually those GPIO-ports are connected to the main MCU of the remote control or of the firing module. This allows interaction with the main MCUs as well as with IO peripherals such as buttons, displays and igniter clips.

The typical use of script entry-points is as follows. The remote control modules process the CSV orchestration scripts. When it decides a *fire* command is required, it sends a ZigBee packet containing a higher-level message to call a specific entry-point on a specific remote module.

Normal device operation. The usual procedure of *normal firing* is as follows. The firing modules are paired with a particular remote control module. Subsequently, firing modules will accept the *arm*, *disarm* and *fire* commands only from the paired remote control module. The pairing is enforced by checking remote control's 802.15.4 *short address* (similar to a MAC address filtering).

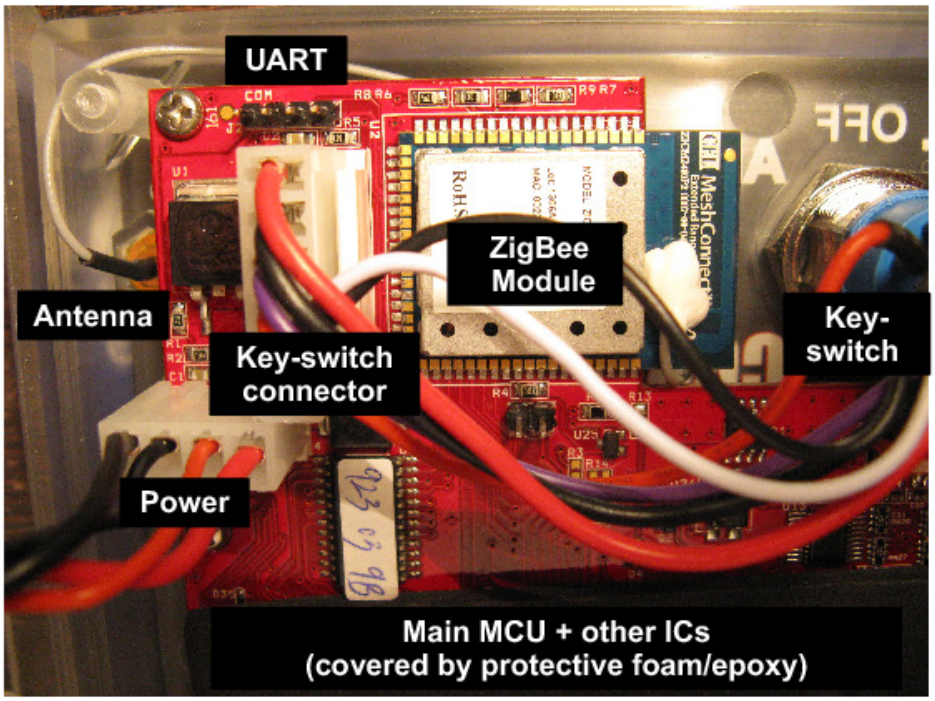

Figure 3.3: Firing module's hardware.

The physical key on all firing modules are turned into *arm* position. The staff departs to the safe regulatory distance to fire the cues. The key on all remote control modules are turned to *on*. The staff confirms everything is safe and ready, and then presses the *arm* button on the remote control, which in turn wirelessly sends a *digital arm* command to firing modules. The firing modules enter a *confirmed arm*, ready for subsequent *fire* command. The staff starts the show by sending, either manually or scripted, *fire* commands to corresponding firing module's cues.

Wireless Attacks

The lack of encryption and mutual unit authentication, opens the system to multiple attacks, in particular sniffing, spoofing and replaying.

We describe a simple attack, yet which we consider as the most dangerous for the fireworks show staff members. The attacker would perform the following sequence of operations in a continuous manner. Eavesdrop the packets (broadcasts, multicasts, node-to-node), from those learn the 802.15.4 addresses of each remote control and firing modules, and learn their corresponding pairing. For each learned pair, the attacker spoofs the remote control's 802.15.4 addresses, spoofs the *digital arm* command to the pair's firing module, and immediately send *fire*

command for all cues once *digital arm* confirmation comes from the firing module. The consequence of this attack is that as soon as the show operator will turn the physical key of a given firing module to *arm* position, it will immediately receive the sequence of *digital arm* and *fire* for all cues. This will fire all the pyrotechnic loads and in the worst case will not allow enough time for the staff to depart to the safe distance. Thus it will defeat the physical key safety and function separation. We *successfully implemented* this attack using components described in Section 3.3.4 and *tested* this attack in practice on the systems we acquired.

Alternatively, an attacker could easily replace default Python functions responsible for firing cues, with arbitrary malicious Python functions. For example, each malicious firing cue function could fire all cues at once instead of firing only it's own cue, thus potentially producing a massive chain explosion. Or it could not fire cues at all or fire them at random, rendering the fireworks show below expectations. Last but not least, an attacker can remotely set random encryption keys on remote nodes. This would result in a denial-of-service for the legitimate user, since her legitimate devices would not be able to communicate with exploited devices anymore. This can definitely ruin a holiday celebration or produce disadvantages to competitors in professional fireworks competitions.

Wireless Attack Implementation

SNAP Stick SS200 [34] It is mainly a firmware programmer for the remote control and firing modules and is based on well-known ATmega128RFA1 chipset from Atmel. Conveniently, using *SNAP Portal*'s utilities, and a special proprietary firmware for it, made available by Synapse as *ATmega128RFA1_Sniffer*, it can be turned into a SNAP-specific 802.15.4 sniffer, where it sniffs and decodes 802.15.4 packets based on Synapse's higher level protocol semantics (e.g., multicasts, broadcasts, peer or multicast RPC calls). We used it to sniff and record the packets between remote control and firing modules during their normal operations. Finally, we also used it to validate our packet injection and replay attacks. If this sniffer received them, then the remote control and firing modules would see our rogue packets. Otherwise we had to fix our injector (regardless the fact that our lower level raw packet sniffer could see them), and then test again sniffed packets and actual devices' behavior.

Wireless Attack Tools

We used the following tools for our experiments.

Goodfet [1] It is an embedded bus adapter for various microcontrollers and radios, additionally proving great open-source support for advanced attacks. It

conveniently provides firmware for TelosB devices to allow sniffing among other functionalities. We tested our attack with this Goodfet firmware running on TelosB.

KillerBee [27] It is a framework and tools for exploiting ZigBee and 802.15.4 networks. It conveniently provides a pre-compiled Goodfet firmware for extra attack functionality. We tested our attack with this Goodfet firmware running on TelosB.

Crossbow's TelosB The sniffer based on SS200 is useful for SNAP protocols and visualization, but it filters out and strips down the packets, hence is largely limiting. We required a lower level raw packet sniffer. We also required an inexpensive and open-source supported approach. TelosB hardware and Goodfet firmware was a perfect fit, so we used them as an additional, much more verbose and raw, sniffer. After learning the SS200 higher level packets for critical commands we correlated them with raw packets recorded by TelosB-Goodfet. Alternatively, a Zigduino [4]could have been used for this task.

Econotag [23] Econotag is a inexpensive and convenient open-source platform for 802.15.4 networks. We assembled sequences of packets instructing to arm and fire sent from the remote control module to the firing module. Finally, we coded an infinite loop of these sequences in a custom firmware. Once plugged, the Econotag successfully performs the attack on a firing module once it's key is turned to $physical\ arm$ position. A Zigduino could have been used for this task as well.

Implementation notes We implemented a simple attack, however it is obvious and trivial to extend the implementation to automatically and continuously sniff new firing modules, and subsequently spoof remote control sequences.

3.3.5 Solutions

Below we summarize a set of recommendations that can dramatically increase the security of the hardware, firmware and wireless communication of the analyzed wireless firing system. With increased security, a safer operation of the entire system can be achieved:

- Provide "factory reset button" to a "factory safe" image and state – this can help reset the wireless chipsets to no encryption state when wireless crypto key (e.g., AES-128) is forgotten.

[4] http://www.logos-electro.com/zigduino/

- In "basic mode" – a clear-text and insecure mode, allow only testing functionality (e.g., identification, communication, continuity).

- In "secure mode" – a mutual authenticated and encrypted mode, allow additional functionality such as *fire* command to igniter clips and firmware upgrade of the both main and wireless MCUs.

- Implement "secure scan" techniques [122] – to allow debugging, testing and restoring of the main MCU and board.

- Remote-code attestation – ensuring, via static or dynamic root of trust, that safety critical code is not tampered with; this could be achieved via hardware and firmware modifications, for example as presented in SMART [104].

- Formal verification – this can dramatically increase security and safety of firmware, hardware and communication protocols.

- Compliance standards and testing – strict compliance testing for both hardware and software, similar to DO-254 and DO-178B respectively. This is also a regulatory issue and should be handled by regulation bodies. We contacted the French association for Standardization (AFNOR), and the vendor contacted the American Pyrotechnics Association (APA) and the Pyrotechnics Guild International (PGI). Unfortunately, this lead to either no response or to an apparent lack of action taken.

3.4 Future Work

On the one hand, we aim at implementing an attack of wireless remote firmware upgrade of the main MCU via the 2.4GHz ZigBee (IEEE 802.15.4) chipsets. This is opposite to the current procedure, where the firmware upgrade is initiated from a USB stick connected locally to the device under attack. Since we have the actual devices under our full control, we also aim at using a dynamic analysis platform for firmware security testing, such as Avatar [203]. An additional aim is to find vulnerabilities in the CSV parser of the remote control to achieve a USB plug-and-exploit proof of concept.

On the other hand, we aim at finding solutions to help this particular category of devices. Solutions not specific to wireless firing systems, include secure firmware upgrades, encrypted and authorized wireless communication channels, secure restore and debug chains. Finally, wireless firing systems specific solutions include secure latency control and secure positioning.

3.5 Summary

We presented vulnerability discovery and exploitation of wireless firing systems in a short amount of time without prior knowledge of such systems. We started with an automated large-scale framework for firmware crawling and analysis (detailed in Chapter 4). In that experiment we employed simple heuristics (e.g., keyword matching) and very simple static analysis. This allowed us to quickly and automatically isolate firmware images of critically-important remote firing systems. We were also able to identify several potential vulnerabilities through both automatic and manual static analysis. These vulnerabilities include unauthenticated firmware upgrade, unauthenticated wireless communications, sniffing and spoofing wireless communications, arbitrary code injection and functionality trigger, temporary denial-of-service. We successfully implemented and tested an unsophisticated attack with potentially devastating consequences.

We conclude that, given the risk presented by their usage, the security of wireless firing systems should be taken very seriously. We also conclude that such systems must be more rigorously certified and regulated. We stress on the necessity and urgency to introduce software and hardware compliance verification similar to DO-178B and DO-254 respectively. We strongly believe these small improvement steps, along with solutions in Section 3.3.5, can definitely help increase the security and safety of such wireless embedded systems.

Last but not least, we discussed the issues with the vendor. A firmware update that is now deployed is addressing most of the security issues. Unfortunately, there are more than 20 vendors of wireless firing systems that may remain vulnerable to similar attacks, in particular some of them do not have a firmware update mechanism.

In this chapter, we demonstrated how security analysis can be performed on a single device by mainly employing manual analysis. While such analysis is very useful to discover serious security issues in embedded devices, this approach does not scale. In the rest of this dissertation we will present techniques to automate some of the steps of the security analysis process for embedded devices.

Chapter 4

A Large Scale Analysis of the Security of Embedded Firmware Images

4.1 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 acquired a broader meaning. The IEEE standard 610.12-1990 [35] 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 [126] or can be maliciously taken over [69, 148]; a home wireless router is found to

have a backdoor [36, 121, 132], just to name a few recent examples. On the one hand, apart from a few projects that targeted specific devices or software versions [82,116,181], 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 re-discovered independently by other researchers [132]. 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.

4.1.1 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 [83]. However, some vulnerabilities are hard to find by just remotely interacting with the running device. Also, 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 contributions within this chapter are 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 4.2), of properly identifying individual firmware images (Challenge B in Section 4.2), of unpacking custom archive formats (Challenge C in Section 4.2), of limiting the required computation resources (Challenge D in Section 4.2), and finally of finding an automated way to confirm the results of the analysis (Challenge E in Section 4.2). While in this chapter 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.

4.1.2 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 39 new vulnerabilities (fixes for some of these are still pending) and to confirm several that were already known [121, 132]. 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 more than 100 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 devices, 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.

4.1.3 Contributions

In summary this chapter 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 39 new CVEs.

4.2 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 often 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 anti-virus 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 [73, 74, 82], *DOS executables* for BIOS, and *ISO images* for hard disk drives [205].

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 [175] and ssdeep [146]) 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 [51, 98]. However, as described in more detail in Section 4.3.4, 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[1]. This simple estimation highlights

[1] 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.

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 [156]. However, since the number of comparisons grows quadratically with the number of elements to compare, this problem quickly becomes computationally infeasible for large image datasets. For example, if 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` [175] 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 this work, in Section 4.3.3 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 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 challenging 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

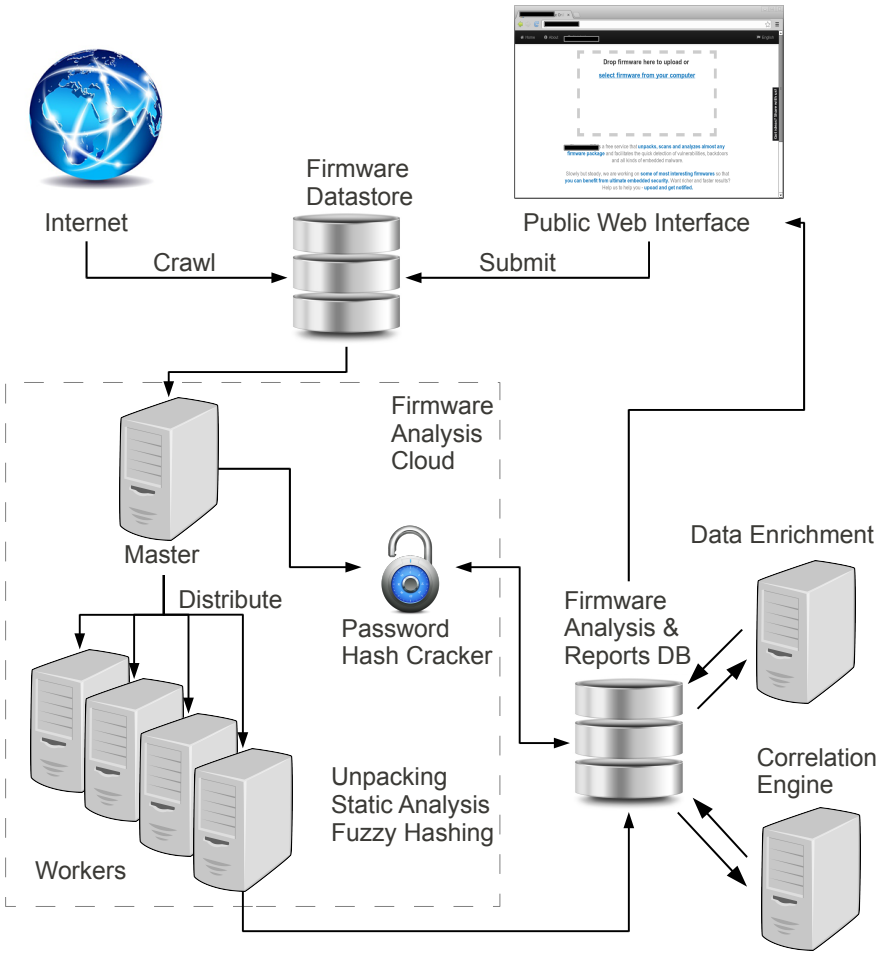

Figure 4.1: Architecture of the entire system.

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.

4.3 Experimental 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 4.2.

4.3.1 Architecture

Figure 4.1 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 4.2), 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.

4.3.2 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 submission 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 [2] 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) [25] 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

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

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).

4.3.3 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 [117], FRAK [81] and Binary Analysis Toolkit (BAT) [192].

The binwalk package is a well known firmware unpacking tool developed by Craig Heffner [117]. 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. [82]. 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 [82]. 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 not supported by our tool at present.

The Binary Analysis Toolkit (BAT), formerly known as GPLtool, was originally designed by Tjaldur software to detect GPL violations [123, 192]. To this end, it recursively extracts files from a firmware blob and matches strings with a database of known strings from GPL projects. Additionally, like binwalk, BAT supports file carving.

Table 4.1 shows a simple comparison of the unpacking performance of each framework on a few samples of firmware images. We chose to use BAT because

Table 4.1: 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 [81]. The additional performance of our framework stems from the many customizations we have incrementally developed over BAT (Figure 4.2).

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%

it is the most complete tool available for our purpose. 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.

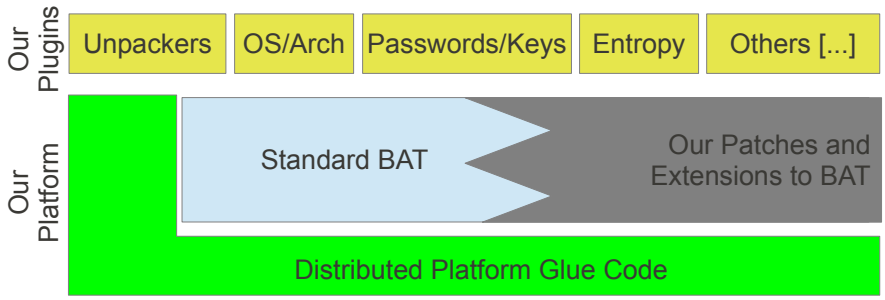

Figure 4.2: Architecture of a single worker node.

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 [152] that run a CUDA-enabled [164] version of *John The Ripper* [165]. 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 firmware files, 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.

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 [187]. Data is synchronized between the repository and the nodes using *rsync (over ssh)* [193].

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 are using 90 such virtual machines to analyze firmware files.

4.3.4 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 firmware and objects inside it.

Shared Credentials and Self-Signed Certificates

Shared credentials (such as hard coded $non\text{-}trivial$ passwords) and shared self-signed certificates are effective in finding strong connections between different firmware images of the same vendor, or even firmware from different vendors. For example, we were able to correlate two brands of CCTV systems based on the fact that they shared a strong 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 4.5.2.

Keywords

Keywords correlation is based on specific strings extracted by our static analysis plugins. In some cases, for example in Section 4.5.1, 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 4.2, 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 4.5.3, 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 [138]. 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 [47] 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. [68] 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 [88] used by BitShred. Second, on each comparison and clustering node we could use the "bins" partitioning approach from French and Casey [47].

4.3.5 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 [155] and GCSE.

Second, we correlate SSL certificates extracted from firmware images to those collected by the ZMap project. ZMap was used in [102] 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 the IP addresses of publicly reachable devices that are running a given firmware image. This provides 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 4.5.2.

4.3.6 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` [201]) to extend BAT, develop new unpackers, create the results analysis platform and run results interpretation.

4.4 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 5.5, 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.

4.4.1 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* [61] 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 to be clearly firmware images. With a confidence of 95%, this translates to a proportion of 34% ($\pm$ 8%) firmware images on our dataset. The manual analysis process took approximately one person-week because the inspection of the extracted files for firmware code is quite tedious.

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 4.3.3, 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.

4.4.2 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

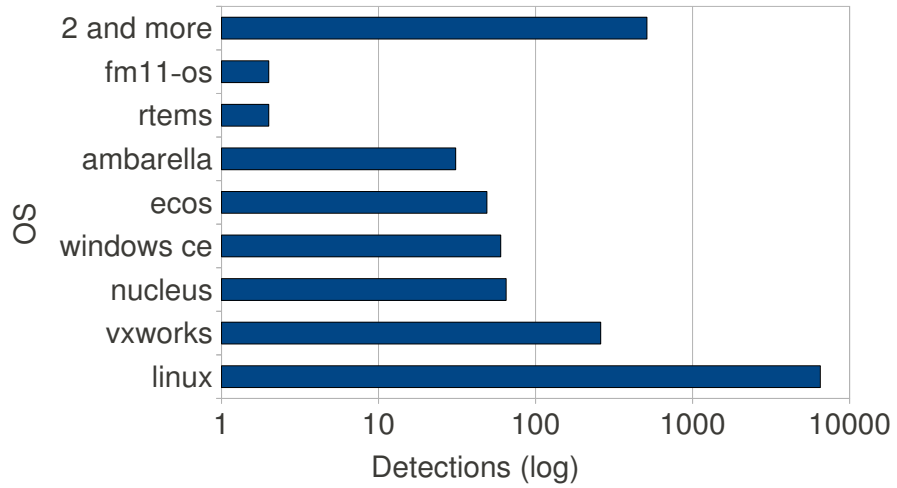

Figure 4.3: OS distribution among firmware images.

reported in Figure 4.3, 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 [119, 120]. 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 [134, 136].

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 [101], 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 [166] 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.

4.5 Case Studies

4.5.1 Backdoors in Plain Sight

Many backdoors in embedded systems have been reported recently, ranging from very simple cases [121] to others that were more difficult to discover [135, 185]. In one famous case [121], 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 [22].

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 [37, 38].

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.

4.5.2 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.

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 [101, 102]. *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 4.4.

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.

4.5.3 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 [202].

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

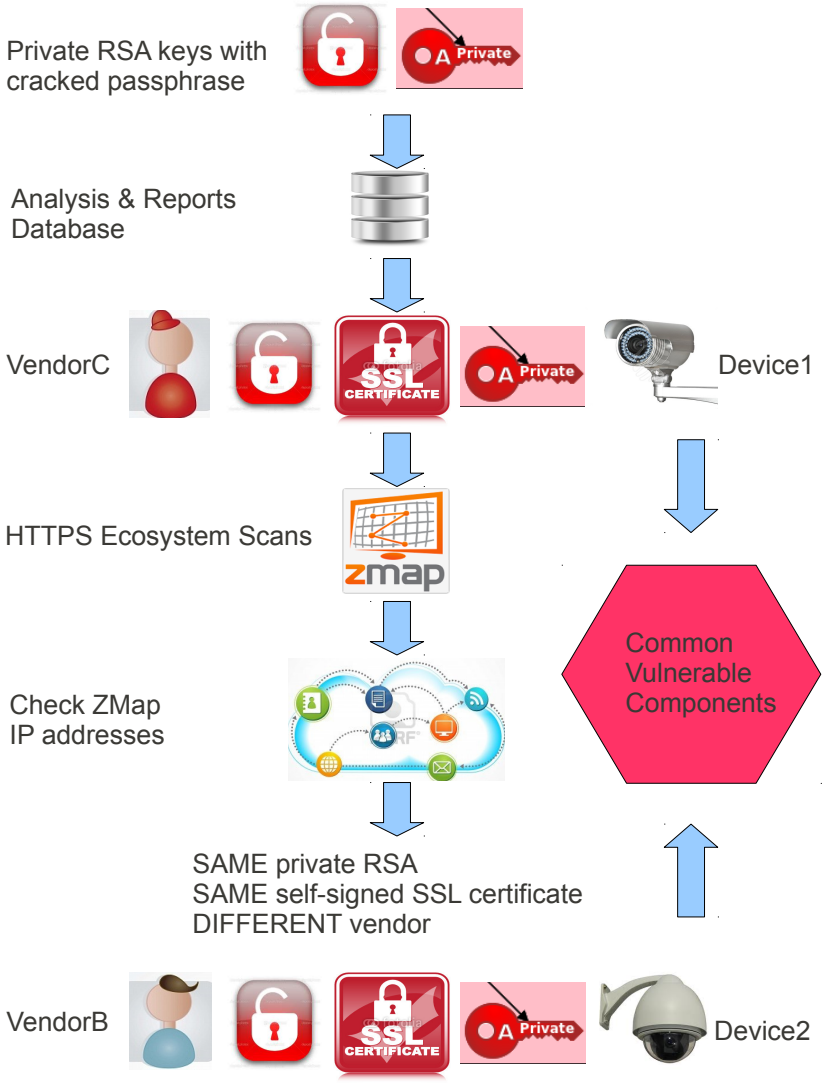

Figure 4.4: Correlation engine and shared self-signed certificates clustering.

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 4.5. 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.

However, such manual vulnerability confirmation does not scale. Therefore, we integrated some of the static analysis tools [16, 44, 85, 105, 139] into our scalable framework. Also, we developed dynamic analysis techniques that scale. In Chapter 5 we show these techniques integrate in our framework and we demonstrate their effectiveness by finding vulnerabilities in real world firmware images.

4.6 Future Work

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 presented in Chapter 3.

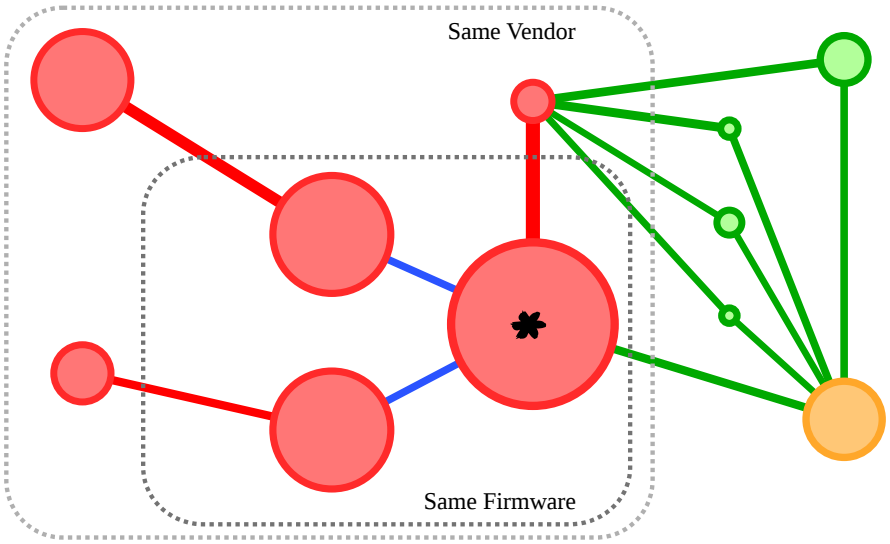

Figure 4.5: Fuzzy hash clustering and vulnerability propagation. A vulnerability was propagated from a *seed file (*)* to other two files from the same firmware and three files from the same vendor (in red) as well as one file from another vendor (in orange). Also four non-vulnerable files (in green) have a strong correlation with vulnerable files. Edge thickness displays the strength of correlation between files.

4.7 Summary

In this chapter we presented a large-scale static analysis of embedded firmware images. 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. The summarized datasets are available at `http://firmware.re/usenixsec14`.

In the following next two chapters we describe several improvements to our framework. In Chapter 5, we attempt to emulate the firmware images by running the unpacked firmware inside the QEMU emulator. We do this to allow a scalable dynamic and static analysis. We show the effectiveness of the improvement by performing scalable analysis of embedded web interfaces within several hundreds of firmware images. Then, in Chapter 6, we apply machine learning to classify and label unknown firmware images. We also use multi-score fusion at the HTTP level to fingerprint embedded online devices. With these improvements, we partially address the "Building a Representative Dataset", "Firmware Identification", "Scalability and Computational Limits", and "Results Confirmation" challenges presented in Section 4.2.

Chapter 5

Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces

5.1 Introduction

During the past few years, embedded devices became more connected forming what is called the Internet of Things (IoT). Such devices are often put online by composition; attaching a communication interface to an existing (insecure) device. Most of these devices lack the user interface of desktop computers (e.g., keyboard, video, mouse), but nevertheless need to be administered. Albeit some devices rely on custom protocols such as "thick" clients or even legacy interfaces (i.e., telnet), the web quickly became the universal "de facto" administration interface. Therefore, the firmware of these devices often embed a web server running from simple to fairly complex web applications. For the rest of this chapter, we will refer to these as *embedded web interfaces*.

It is well known that making secure web applications is a difficult task. In particular, researchers showed that more than 70% of vulnerabilities are hosted in the (web) application layer [171]. Attackers, who are familiar with this fact, use a variety of techniques to exploit web applications. Well known vulnerabilities, such as SQL injection [62] or Cross Site Scripting (XSS) [199], are still frequently exploited and constitute a significant portion of the vulnerabilities discovered each year [71]. Additionally, vulnerabilities such as Cross Site Request Forgery (CSRF) [45], command injection [188], and HTTP response splitting [142] are also quite often present in web applications.

Given such a track record of security problems in both embedded systems and web applications, it is natural to expect the worse from embedded web interfaces.

However, as we will discuss, those vulnerabilities are not easy to discover, analyze and confirm.

Analysis of embedded web interfaces. While there are solutions that can be used during the design phase of the software [128, 153, 178, 179], it is also important to discover and patch existing vulnerabilities before they are found and abused "in the wild" by the attackers. One way to do so, is to use a "white box" approach, using static analysis of the source code [44, 85, 93, 140]. Another technique is dynamic analysis, where the web interface is typically exercised against a number of known attack patterns [48, 58].

Unfortunately, those tools are either inefficient or difficult to use for detecting vulnerabilities of embedded web servers [48, 112]. Performing static analysis on embedded web interfaces is a rather simple task once the firmware has been unpacked. However, one main limitation of this approach is that the web interfaces often rely on various languages (e.g., PHP, CGIs, custom server-side languages), but the static analysis tools are usually designed for a particular one. In addition to that, many static analysis tools are merely "glorified greps" and have a large number of False Positives (FP), which makes them problematic to reliably use in a large scale study. On the other hand, dynamic analysis tools [110, 130] are more generic as they are less sensitive to the server-side language used. Nevertheless, they require the web interface to be functional. Unfortunately, it is difficult to create an environment that can perfectly emulate firmware images for various devices based on a variety of computing architectures and hardware designs.

Scalable dynamic analysis of embedded web interfaces. The easiest way to preform dynamic analysis is to perform it on a live device. However, acquiring devices to dynamically analyze them is not scalable. Also it is ethically questionable, if not illegal, to test devices one does not own (e.g., devices on the Internet). Another option would be to extract the web interface files from a device and load them in a test environment, like an Apache web server. Unfortunately, a large majority of the embedded web interfaces use native CGIs, bindings to local architecture-dependent tools or custom web server features which cannot be *easily* reproduced in a different environment (Section 5.2.4).

Emulating the firmware is an elegant method to perform dynamic analysis of a system, since it does not require the physical device to be present and can be completely performed in a controlled environment. Sadly, emulation of unknown devices is not easy because an embedded firmware expects specific hardware to be fully present, such as peripherals or memory layouts. Previous attempts were made at improving emulation of firmware images by forwarding I/O to the hardware [203], which applies to many kind of embedded systems, even the monolithic firmware images. In a different approach [141], Linux based embedded systems are emulated with a custom kernel that forwards `ioctl` requests to the

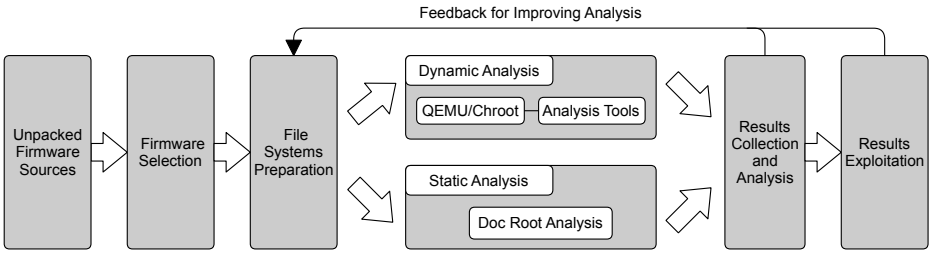

Figure 5.1: Overview of the analysis framework.

embedded device that runs the original kernel. Those techniques achieve a rather good emulation, but require the presence of the original device, which does not scale. We observe that, in Linux based embedded systems, the interaction with the hardware is usually performed from the kernel. Moreover, web interfaces often do not interact with the hardware or this interaction is indirect.

In this chapter, we propose a partial emulation of firmware images by replacing their kernel with a stock kernel (targeting the same architecture) and emulate the whole userland of the firmware using a hypervisor, such as QEMU [106]. We `chroot` the unpacked firmware and start the `init` program, the init scripts or, sometimes, directly the web server. Once (and if) the web server is up and operational, we use dynamic analysis tools to discover vulnerabilities in the system. This approach has the advantage to be rather automated and generic, however, a complete emulation is very slow and some firmware images and tools do not run properly.

5.1.1 Overview of our Approach

In order to perform scalable security testing of embedded web interfaces we developed a framework for automated analysis (Figure 5.1). We started our analysis with a dataset of 1925 unpacked firmware images [1] that contain embedded web interfaces. Then, for each unpacked firmware we identify any potential web document root present inside the firmware. At this point we make a pass with static analysis tools on the web document root. Next, we emulate the firmware images. Once (and if) the web server is up and operational, the dynamic analysis phase is performed. Finally, we analyze the results and perform manual analysis wherever needed.

[1] We focused mainly on Linux-based firmware images. Linux-based firmware images are in general well structured and documented, therefore they are easier to unpack, analyse and emulate. However, our approach can be easily extended in the future to other types of firmware, including monolithic ones.

5.1.2 Contributions

In summary, we make the following main contributions:

- We perform the first large scale, comprehensive, security study on embedded web interfaces, leveraging multiple techniques and state of the art tools.
- We shed light on a previously unstudied part of firmware and discover serious vulnerabilities in a wide spectrum of embedded devices.
- We propose an efficient methodology and we develop a scalable framework to address detection of web vulnerabilities in embedded devices.
- We enable a testbed for further advanced security research on firmware of embedded systems.

5.2 Exploring Techniques to Analyze Web Interfaces of Firmware Images

We evaluated several options for performing static and dynamic analysis on web interfaces of embedded systems. Indeed, many solutions are possible, but not all fit our needs. We summarize here the different possibilities and motivate our choices.

5.2.1 Static Analysis

There are many practical advantages to static analysis tools; they are often automated and do not require setting up too complex test environments. In general, they only need the source code (or application) to be provided to generate an analysis report. It is also relatively easy to plug new static analysis tools for increased coverage or wider support of file formats and source code languages. Finally, as result of all the above, such tools are scalable and are easy to automate.

However, static analysis techniques have well understood limitations. On the one hand, they cannot find all the vulnerabilities, which results in a number of missed vulnerabilities, i.e., *False Negatives (FN)*. On the other hand, they also alert on non-vulnerabilities, i.e., *False Positives (FP)*. When a static analysis tool is used to improve or guide a manual analysis the false positives can be discarded quickly. However, when performing a large scale study a too high FP rate is very problematic as manually discarding them becomes infeasible.

Additionally, the firmware within embedded devices use technologies for which security static analysis tools often do not exist (e.g., `perl`, `lua`, `haserl`, `binary CGIs`). On the contrary, there exist a number of static analysis tools

for PHP [85, 139], mainly due to the fact that PHP is the most popular server-side programming language for the web [2]. Unfortunately, it appears from our dataset that only a portion of embedded web interfaces actually use PHP in their server-side. This is not really a surprise as PHP is not designed for embedded systems. We nevertheless examined these cases, by using *RIPS* [85] and provide results of this analysis in Section 5.5.2.

Finally, *binary static analysis* can be applied to binary CGIs to find vulnerabilities such as buffer overflows, (remote) code executions, command injections. One possible solution is to use tools like `Firmalice` [184] or `WEASEL` [181]. However, it is challenging to find static binary analysis tools that are completely automated and which cover the variety of architectures present in our dataset (`arm, armel, mips, mipsel, powerpc, cris, m68k, blackfin`).

Static analysis tools. There exist many static analysis tools [3, 4]. For instance, *RIPS* is a state of the art static analysis tool for PHP source code [5]. It implements a context-sensitive, intra- and inter- procedural data flow analysis. It uses tainting and basic block, function, and file summaries for and efficient, backwards-directed data flow analysis to discover a wide range of web vulnerability categories. We found it to provide very good results.

There are several other tools that provide static analysis. Unfortunately, we excluded them from our static analysis infrastructure for various reasons. Some tools like *RATS* [6], *VisualCodeGrepper* or *Yasca* [7] support some web scripting languages, but are performing only rough analysis of the source code warning about usage of sensitive APIs such as `system()` or `exec()`. Those tools do not perform semantic or Abstract Syntax Tree (AST) analysis, and hence they yield a lot of false positives and are not suitable for a large scale analysis. Other tools, such as *Pixy*, are not available nor maintained. Finally, there are many commercial static analyzers, however, it was difficult to get access to them, in particular due to their high cost. We therefore did not experiment with any of them. We expect that such tools would be easy to integrate in our framework; while they would not fundamentally change our approach they would certainly improve our results.

5.2.2 Dynamic Analysis

Dynamic analysis—an analysis that relies on testing an application by running it—has many benefits. First, dynamic analysis of web interfaces is mostly independent from the server-side technology used. For instance, the same tool can test interfaces that are implemented in PHP, native CGIs or custom web scripting engines. This allows to greatly improve the coverage of tests and to examine interfaces with the same tools independently of their technology. Second, it can be used to confirm vulnerabilities found in the static analysis phase. There are

many dynamic analysis tools for security testing of web applications. Bau et. al. [48] performed a detailed evaluation of such web vulnerability testing tools.

One big advantage of dynamic analysis tools is that they have more accurate analysis results and perform a better validation of the reported vulnerabilities. They identify problems in a target environment and can do so even when there is no access to the source code. Unfortunately, dynamic analysis tools often require a lot of effort to setup (e.g., environment setup), and sometimes also require customization such as adding new vulnerability modules for scanning and testing.

For our framework we selected tools that are open source so that we can easily adapt and integrate them in our framework and fix their defects when needed. Based on this we used *Arachni* [8], *Zed Attack Proxy (ZAP)* [9] and *w3af* [10]. *Arachni* is an open source framework, written in Ruby, which is designed for penetration testing of web applications. We found it to provide very good results. *ZAP* is a Java-based integrated penetration testing tool for web applications from OWASP. It aims to be easy to use by a broad audience. We found it to provide mixed level of results. *w3af*, or Web Application Attack and Audit Framework, is a Python framework designed to help developers secure web applications by finding and exploiting a broad range of web application vulnerabilities. We found it to provide results that are not as good as those of *ZAP* and *Arachni*.

5.2.3 Limitations of Analysis Tools

Inherent limitations. Tools in general are designed for very specific purpose, with certain assumptions in mind. One example is that there are high number of FPs and FNs that are the direct consequence of the vulnerability finding tools we rely on. Another example is the implementation of *OS command injections* in most of the tools. Most of these vulnerabilities are often missed because such flaws are often hard to discover via automated testing [11]. Tools usually try to inject commands such as `ping <ip>` or `cat /etc/passwd`. These tests implicitly assume that the networking is functional and that the firmware has the `ping` utility or that the output from commands are not filtered by the web application. To overcome these limitations we take advantage of our "white box" approach (see Section 5.3.4).

Defects in analysis tools. The development effort of our framework went much beyond plugging the right tool in the right place. Indeed, we had to extend many tools and we encountered many defects in them. Those issues were severely impacting the success rate of the vulnerability discovery. We were able to fix many of them but this required a significant engineering effort. Fixing bugs proved necessary to obtain good results. While there are still defects to be fixed, this shows that better web application analysis tools are still needed, in particular for embedded devices.

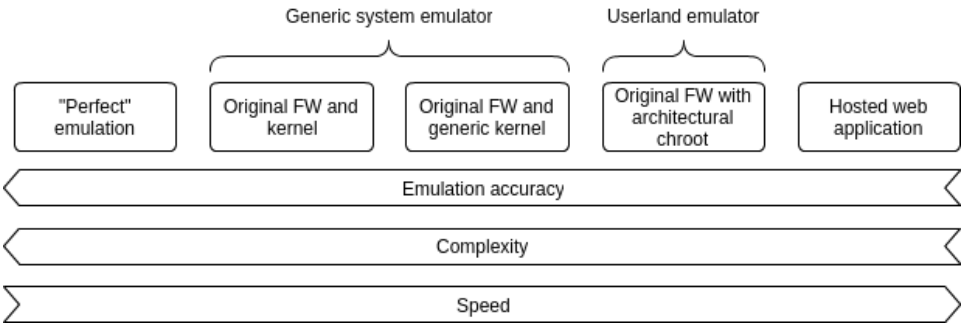

Figure 5.2: Various possible options to launch a web interface: from perfect emulation of a hardware platform to hosting the web interface. Arrows are indicative of a general trend, actual evolution of the properties may not be linear.

5.2.4 Running Web Interfaces

Dynamic analysis of web applications require a function web interface. There are different ways to launch the web interface that is present in the firmware of an embedded system, however, none of them are perfect. Some methods are very accurate but infeasible in our setup, such as emulating the firmware in a perfect emulator (which is not available). Other methods are much less accurate, like extracting the web application files and serving them from a generic web server. Therefore, we evaluated different possibilities and describe their advantages and drawbacks (summarized in Figure 5.2).

Hosting Web Interfaces Non-Natively

A straightforward way to launch a web interface from a firmware is to extract it (i.e., the document root) and launch it under a web server on an analysis environment, without trying to emulate the original web server and firmware. For this, we need to identify the web server used by the firmware (e.g., `boa`, `minihttpd`, `thttpd`, `lighttpd`, `apache`) and to locate its original configuration. Then the web application is located (described in Section 5.3.2), extracted and "transplanted" to the *hosting environment*. The hosting environment is set up using the same web server (or a very similar version) and the configuration files extracted from the firmware. The main advantage of this technique is that it does not require emulation, which dramatically simplifies the deployment and thus, it is easy to automate.

However, this approach has many limitations. For example, it is not possible to handle platform dependent binaries and CGIs. We analyzed the document roots within the 1580 firmware candidates for emulation and found that 57% out of those were using binary CGIs or were in some way bound to the platform[2]. In

[2]This is a lower bound as we did not count web scripts calling local system utilities, e.g.,

addition to this, the firmware images often use either customized web servers, or versions which are not available on normal systems (e.g., `uClinux` has a custom version of the common `boa` web server). Those often support options which are nontrivial to reproduce under a normal system without profoundly modifying the web server used. As a consequence, the chances of properly emulating the web interfaces using this technique is very low and after careful evaluation on a few firmware samples we decided not to use it in our framework.

Firmware and Web Interface Emulation

To emulate a firmware (or part of it) we need to detect the CPU architecture it is intended to execute on. Also, depending on the chosen emulation method, a generic kernel and a file system for that architecture could be required. Detecting the CPU architecture of each root filesystem is not trivial. For example, some firmware packages contain files for various architectures (e.g., ARM and MIPS). Sometimes, vendors package two different firmware blobs into a single firmware update package. The firmware installer then picks the right architecture during the upgrade based on the detected hardware. In such cases, we try to emulate this filesystem with each detected architecture. We detect the architecture of each executable in a firmware using ELF headers or statistical opcode distribution for raw binaries. We then decide on the architecture by counting the number of architecture specific binaries it contains. Once we detected the right architecture, we use the QEMU emulator for that particular architecture. There are different possibilities for emulating the firmware images, which we now compare.

Perfect emulation. Ideally we would have a complete firmware (including the bootloader, kernel,. . .) and a QEMU setup that perfectly emulates the hardware for which the firmware was designed. However, QEMU only emulates few platforms for each CPU architecture. Moreover, perfectly emulating unknown hardware is impossible, especially considering that hardware devices can be arbitrarily complex. In addition to this, hardware in embedded devices is often custom and its documentation is in general not available. It is therefore infeasible to adapt the emulator, and even less to apply this at a large scale.

Original kernel and original filesystem on a generic emulator. Reusing the kernel of the firmware could lead to a quite accurate emulation, in particular because it may export interfaces for some custom devices that are needed to properly emulate the system. Unfortunately, kernels for embedded systems are often customized and hence, do not support a wide range of peripherals. Therefore, using the original kernel is unlikely work very well. Furthermore, the original

using the `system()` call.

kernel is often missing from the firmware image under analysis. For instance, we found only 107 kernels out of 1925 original firmware images. Consequently, for most of the embedded systems, the kernels have to be extracted from the device (e.g., via JTAG, flash dumping). We therefore did not try to use the original kernels.

Generic kernel and original filesystem on a generic emulator. We can easily find or build a complete generic kernel for the architecture of a device[3]. Such a kernel will then boot properly on the generic emulator. The lack of the original kernel can reduce the accuracy of the emulation. However, we estimate this limitation does not affect significantly the web interface emulation and hence, the coverage of the dynamic analysis. Another drawback is the raw usage of the original firmware filesystem. As these firmware images are built differently by different vendors for varying devices, we cannot fully rely on the presence of required utilities, e.g., `SSH, SCP, netstat`. Without such basic tools, it becomes problematic to do automated management of the emulated hosts. We have evaluated this technique on a few selected firmware instances. Though we could build and install generic support tools, there are more elegant solutions which we are going to present.

Generic kernel and a generic filesystem, chrooting the firmware to analyze. We first boot a generic Debian Linux system and generic kernel inside the QEMU emulator. Once this generic environment is up and operational, we copy the root filesystem of the firmware into the emulated environment. We then chroot to the firmware's filesystem and execute (inside the chroot) the shell (e.g., `/bin/sh`) or the init binary (e.g., `/sbin/init`). Finally, in the chroot environment we start the web server's binary along with the web interface document root and web configuration.

The main advantage of this approach is that it can provide almost complete emulation of the web interfaces of the embedded devices and is scalable. There are few drawbacks of this approach. First, emulating the system is not very fast; emulation is one order of magnitude slower than native execution. Additionally, the emulator environment setup and cleanup introduces a significant overhead. This approach also limits the number of platforms to those supported by the `Debian Ports` project. Luckily, the Debian project supports many common embedded architectures (e.g., `arm, mips, powerpc`).

Unfortunately, with this approach we cannot fully emulate the peripherals and specific kernel modules of the embedded devices. However, few firmware images and a limited part of embedded web interfaces actually interact directly with the

[3]We used the pre-compiled Debian Squeeze kernels and systems from https://people.debian.org/~aurel32/qemu/

peripherals. One such example is a web page that performs a firmware upgrade which in turn requires access to `flash` or `NVRAM` memory peripherals.

In summary, this is the approach that we found to be most scalable and that provided the best results in starting the web interfaces for analysis. At the same time, this approach is a balanced trade-off between the emulation accuracy, complexity and speed (see Figure 5.2).

Architectural chroot. One way to improve the performance and emulation management aspects of our framework is by using *architectural chroot* [12], or *QEMU static chroot*. This technique uses `chroot` to emulate an environment for architectures other than the architecture of the running host itself. This basically relies on the Linux kernel's ability to call an interpreter to execute an ELF executable for a foreign architecture. Registering the userland QEMU as an interpreter allows to execute `arm` Linux ELF executables on an `x86_64` Linux system. Our approach is to do userland emulation only with architectural root (Figure 5.4). We have successfully tested this technique on a few firmware instances. This approach is fast and easy to manage (compared to the system QEMU emulation), however, it is quite fragile and requires significant additional work to be more reliable. In addition, while this approach has the advantage of improving emulation speed, it is unlikely to improve the number of firmware packages we can emulate in the end. Therefore, we did not use this technique in our experiments. We plan to explore this direction in more details in our future work.

5.3 Analysis Framework Details

In order to perform a large scale and automatic analysis of firmware images we designed a framework to process and analyze them (Figure 5.1). First, we obtain a set of unpacked firmware images, analyze and filter them (Section 5.3.1). Then we perform some pre-processing of the selected unpacked files. For instance, some firmware images are incompletely unpacked or the location of the document root is not obvious (Section 5.3.2). We then perform the static and dynamic analyses (Section 5.3.3). Finally, we collect and analyze the reported vulnerabilities (Section 5.3.4), and exploit these results (Section 5.3.5).

5.3.1 Firmware Selection

The firmware selection works as follows. First, we select the firmware images that are successfully unpacked and are Linux-based systems which we can natively emulate and chroot to (see Section 5.3.2). Second, we choose firmware instances that clearly contain web server binaries (e.g., `httpd, lighttpd`) and typical

configuration files (e.g., `lighttpd.conf, boa.conf`). In addition to these, we select firmware images that include server side or client side code related to web interfaces (e.g., `HTML, JavaScript, PHP, Perl`). Our dataset is detailed in Section 5.4.

5.3.2 Filesystem Preparation

To emulate a firmware the emulator requires its root filesystem. In the best case the unpacked firmware directly contains the root filesystem. However, in many cases the firmware images are packed in different and complex ways. For example, a firmware can contain two root filesystems, one for the upgrade and one for the factory restore, or it can be packed in multiple layers of archives along with other resources. For these reasons, we first need to detect the potential candidates for root filesystems. We achieve this by searching for key directories (e.g., `/bin/, /sbin/, /etc/, /usr/`) and for key files (e.g., `/init, /linuxrc, /bin/busybox, /bin/sh, /bin/bash, /bin/dash`). Once we detect such key files and folders relative to a directory within the unpacked firmware, we select that particular directory as the *root filesystem* point. There are also cases when it is hard or impossible to detect the root filesystem. A possible reason for this is that some firmware updates are just partial and do not provide a complete system. For each detected root filesystem, we extract it from the unpacked directory tree and pack it as a standalone root filesystem ready for emulation.

Unpacking firmware images can produce "broken" root filesystems and thus, we need to fix them. In addition to this, sometimes the initialization scripts do not explicitly start (or fail to start) the webserver. Therefore, in such cases in order to start the web server within the root filesystem, we need to detect the web server type, its configuration, and the document root. For these reasons, we have to use heuristics on the candidate root filesystems and apply transformations before we can use them for emulation and analysis.

Filesystem Sanitization

Unpacking firmware packages is not perfect. First, unpacking tools sometimes have defects. Second, some firmware images have to be unpacked using an imperfect "brute force" approach (see Chapter 4). Finally, some vendors customize archives or filesystem formats. For instance, we observed a case in which the symlinks are incorrectly unpacked because they are represented as text files that contain the target of the link (e.g., the symbolic link `/usr/bin/boa -> /bin/busybox` is represented with a text file named `/usr/bin/boa` that contains the string `/bin/busybox`). All these lead to an incorrect unpacking, thus the unpacked firmware image differ from the filesystem representation intended to be on the device. This reduces the chances of successful emulation and therefore we need a sanitization phase.

This sanitization phase is performed by scripts that traverse unpacked firmware filesystems and fix such problems. Sometimes, there are multiple ways to fix a single unpacked firmware. This results in multiple root filesystems to be submitted for emulation increasing our chances of proper emulation of a given firmware. Implementing these heuristics added a 13% processing overhead. At the same time, it allowed us to increase the successful emulations by 2% and the successful web server launches by 11%.

Web Server Configuration

We inspect the firmware files and locate the web server binaries (`httpd`, `boa`, `lighttpd`, `thttpd`, `minihttpd`, `webs`, `goahead`). We also locate their related configuration files (e.g., `boa.conf`, `lighttpd.conf`). The path of the web server and its configuration file is sufficient to start the web server using a command such as `/bin/boa -f /etc/boa/boa.conf` (a real example from our dataset). Additionally, we extract important settings from the configuration files (e.g., the document root).

Web Server Parameters

Sometimes, we miss important parameters which are required to properly start the web server, such as the document root path or the CGI path. Often this happens because of a missing configuration file (e.g., partial firmware update) or because the parameters are supplied via the command line from a script which is not available. In these cases, we experiment with all the potential document roots of the firmware. To find a potential document root (within the root filesystem) we first search for index files (e.g., `index.html`, `default.html`) with possible file extensions (`HTML, SHTML, PHP, ASP, CGI`). Then, we build a set of *longest common prefix directories* of these files. This can result in multiple document root directories, for example a second document root can be found in a recovery partition. Once we discover the document roots, we prepare the possible commands to start the web server. By using this approach, we increase the chances to succeed in starting a working web server.

Web Server Sitemap

We also build an optimized site map for each such document root directory. The site maps are very useful to be provided as an input to the dynamic analysis tools. Indeed, one of the well known limitations of the dynamic analysis tools is that they need to crawl the web application to discover the pages to check, which can be inefficient and reduces the coverage [95]. Providing the site map mostly solve this problem. Thus, we instruct the tools to restrict their analysis to

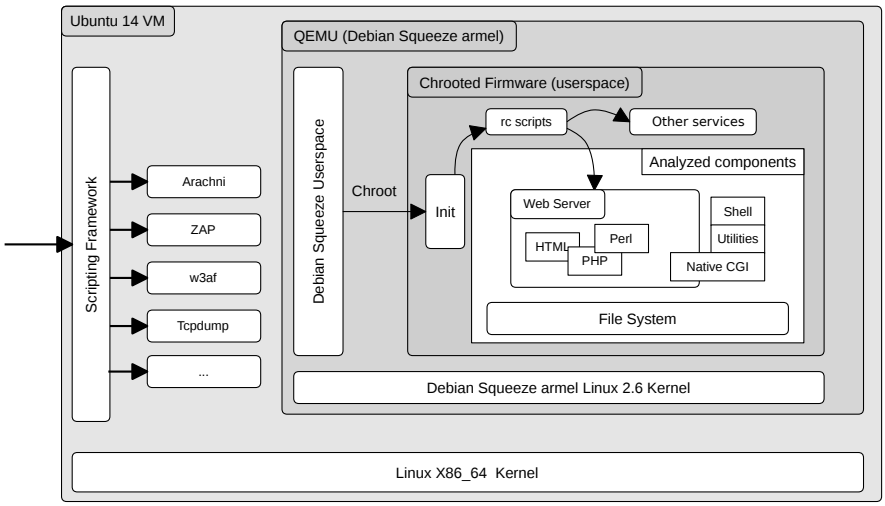

Figure 5.3: Overview of one analysis environment for Linux armel with a 2.6 kernel.

the supplied site map and we do this for multiple reasons. First, it significantly reduces the time required to complete the dynamic analysis. No time is wasted to analyze uninteresting files, such as image files, or to crawl the web application. Second, it reduces the chances for the web interface or the emulator to crash by limiting the resource load, e.g., number of requested files. Third, it increases the chances that the files which were reported as vulnerable by static analysis will also undergo dynamic analysis.

5.3.3 Analysis Phase

Once the filesystems are prepared, we emulate each of them in an analysis virtual machine where dynamic testing is performed (Figure 5.3 and Section 5.2.2). We also submit the document roots to the static analyzers (Section 5.2.1). This phase is completely automated and scalable as each of the firmware images can be analyzed independently.

5.3.4 Results Collection and Analysis

After dynamic and static analysis phases are completed, we obtain the analysis reports from the security analysis tools in XML format. We also collect several logs that can help us make further analysis as well as improve our framework. These are typically required to debug the analysis tools or our emulation environment. For instance, we collect SSH communication logs with the emulator host, changes in the firmware filesystem, and capture the network traffic of the interaction with the web interface.

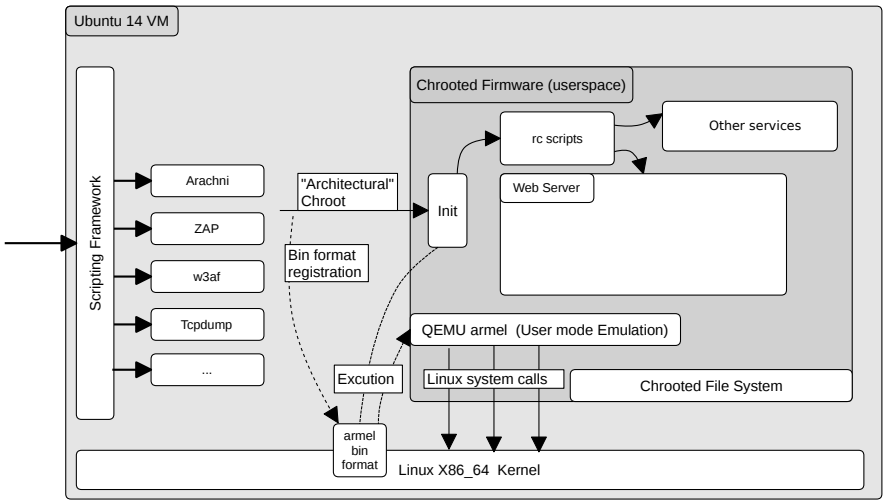

Figure 5.4: Architectural chroot analysis setup.

File systems changes. We capture a snapshot of the emulated filesystem at several different points in time. We do this (i) before starting the emulation, (ii) after emulation is started, and (iii) after dynamic analysis is completed. Then, we perform a filesystem `diff` among these snapshots. Interesting changes are included in both log files and new files. Log files are interesting to collect in case a manual investigation is needed. New files can be the consequence of a *OS command injection* or more generally of a *remote code execution (RCE)* vulnerability triggered during the dynamic analysis phase. This often occurs when dynamic testing tools try to inject commands (e.g., `touch <filename>`). Sometimes, the command injection can be successful but is not detected by the analysis tools. However, it is easy to detect such cases with the filesystem diff.

Capturing communications. Performing dynamic analysis involves a lot of input and output data between the (emulated) device and the dynamic analysis tool. It makes sense to capture both the raw input and the output communication. In case problems arise during testing, this capture can add an increased trace of accountability. However, there are few more reasons to capture this raw communication.

A successful *OS command injection* can go undetected by the tools. Also, such a vulnerability can be difficult to verify, even in a "white box" testing approach (Section 5.2.3). After the testing it can be discovered that a command injection was in fact successfully triggered and accomplished. In such a case, it is needed to rewind through all HTTP transactions to find the input triggering the particular vulnerability and we can then look for incriminating inputs and parameters (e.g., a `touch` command).

The test tools often behave like fuzzers as they try many malformed inputs one after the other. Because of this, a detected vulnerability may not be a direct result of the last input. For example, it can be a result of the combination of several previous inputs. It is therefore important to recover all these previous inputs in order to reproduce the vulnerability. We use `tcpdump` to capture the communication, which we store in a the dump file. Then we use `justniffer` [13] to analyze the HTTP streams and to recover the exact URLs and their parameters.

5.3.5 Results Exploitation

After collecting all the details of analysis phase, we perform several steps to exploit these results. First, we validate the high impact vulnerabilities by hand and try to create a proof-of-concept exploit. This could be automated in the future, as this was done for other fields of vulnerability research [42]. Unfortunately, none of the tools we currently use provide such a functionality. Additionally, from the static analysis reports we manually select the high impact vulnerabilities (e.g., command injection, XSS) and the files they impact. We then use these to explicitly drive the dynamic analysis tools and aim mainly at two things: (i) get the dynamic analysis tools to find the vulnerabilities they missed (if they did) and (ii) find the bugs or limitations that prevented the dynamic tools to discover the vulnerability in the first place. In addition to this, sometimes we do selective manual analysis on source code files of the embedded web interface. For example, a source code file (e.g., PHP) could be reported by the dynamic or static analysis tools to have a large number of high-impact vulnerabilities. In such cases, we manually check the source code files to ensure those reports are not false positives or bugs in the tools, and also we try to find other vulnerabilities during manual inspection. Even though manual analysis does not scale, it can help uncover additional nontrivial vulnerabilities (see Table 5.7). Finally, we summarize all our findings in vulnerability reports to be submitted as CVEs.

5.4 Dataset

We started this study with an initial dataset of unpacked firmware images that we previously discussed in Chapter 4. From this initial dataset, we selected firmware images based on several properties. First, we chose the firmware instances which were successfully unpacked and which were Linux-based embedded systems. These were the systems which were likely the simpler to natively emulate and chroot. Then, we selected firmware instances that clearly contained a web server binary (e.g., `httpd, lighttpd`) and typical configuration files (e.g., `lighttpd.conf, boa.conf`). In addition to these, we also chose firmware images that included server-side or client-side code correlated to web interfaces (e.g., `HTML, JavaScript, PHP`).

Table 5.1: Firmware counts at various phases of the dynamic analysis of embedded web interfaces.

Dataset phase	# of FWs (unique)	# of root FS
Original firmware	1925	–
Web server emulation candidates	1580	1754
Improved by heuristics	1580	1982
Emulation OK	488	–
Web server OK	246	–

Table 5.2: Distribution of architectures and their emulation success rates.

Arch.	Original FWs	Emulation OK	Web server OK
arm	32%	53%	55%
mips	18%	21%	17%
mipsel	17%	26%	28%
bflt	5%	no support yet	no support yet
cris	4%	no support yet	no support yet
powerpc	3%	no support yet	no support yet
others	21%	no support yet	no support yet
Total	**1925**	**488**	**246**

Challenges and Limitations Inevitably, our dataset and the heuristics we apply lead to a bias, as explained in Chapter 4. First, it almost only contains firmware images that are publicly available online. Second, Linux based devices only account for a portion of all embedded systems. Finally, there are firmware images running as monolithic software or embedding web servers we currently do not detect or support. We are aware of this bias and the results herein should be interpreted without generalizing them to all embedded systems. In essence, these choices were needed to perform this study and it will be an interesting future work to extend the study to more diverse firmware images. However, adapting our framework and techniques to embedded devices that have their firmware based on systems with a clear separation of bootloader, kernel space, user space, and filesystems (e.g., VxWorks, WinCE, QNX) should be relatively easy.

5.5 Results and Case Studies

5.5.1 Overview of Discovered Vulnerabilities

Our automated system performed both static and dynamic analysis of embedded web interfaces inside 1925 firmware images. We discovered at least 244 high impact vulnerabilities, and still have to analyze and confirm static analysis re-

Table 5.3: Distribution of web servers types among the 246 instances which successfully started a web server.

Web server	% among started web servers
minihttpd	37%
lighttpd	30%
boa	4%
thttpd	3%
others	26%

Table 5.4: Distribution of web technologies within the 246 instances which started a web server.

Web interface contains	% of started web servers
HTML	98%
CGI	57%
PHP	2%
perl	3%
POSIX shell	11%

ports of some more 9046 possible vulnerabilities, overall affecting 185 firmware packages from 13 vendors.

5.5.2 Static Analysis Vulnerabilities

PHP is one of the most used server-side web programming languages [96]. Over the past years, many researchers focused on investigating vulnerabilities in PHP applications and creating static analysis tools [85, 139]. However, to the best of our knowledge, we are the first to study the prevalence of PHP in embedded web interfaces and their security. We extracted and analyzed the PHP source code within our dataset. RIPS reported 145 unique firmware packages to contain at least one vulnerability and a total of 9046 reported issues. The detailed breakdown is presented in Table 5.5. We observe that cross-site scripting and file manipulation constitute the majority of the discovered vulnerabilities, while command execution (one of the most serious vulnerability class) ranks third.

5.5.3 Dynamic Analysis Vulnerabilities

Our framework was able to perform security testing on 246 systems, and the general results are presented in Table 5.7. In particular, we discovered 21 firmware packages which are prone to command injection. While this seems a small number of vulnerabilities, their impact is significant as there is potentially a very high number of devices running these firmware images, and users affected. These

Table 5.5: Distribution of PHP vulnerabilities reported by RIPS static analysis. NOTE: For TP, FP, FN rates of each vulnerability type see Table *Evaluation results for popular real-world applications* in [85].

Vulnerability type	# of issues	# of affected FWs
Cross-site scripting	5000	143
File manipulation	1129	98
Command execution	938	41
File inclusion	513	40
File disclosure	461	87
SQL injection	442	10
Possible flow control	171	56
Code execution	141	21
HTTP response splitting	127	27
Unserialize	119	15
POP gadgets	4	4
HTTP header injection	1	1
Total	**9046**	**145 (unique)**

serious vulnerabilities are affecting devices such as SOHO routers, CCTV cameras, smaller WiFi devices (e.g., SD-cards). Correlating these firmware images to populations of online devices is left for future work. In Chapter 4 we have shown this to be possible, for example, using Shodan [155] or ZMap [102].

Additionally, we found 32 firmware packages affected by XSS and 37 vulnerable to CSRF. Even though XSS and CSRF vulnerabilities are usually not considered to be critical vulnerabilities, they can have a high impact. For example, Bencsáth et al. [52] used a Cross-Channel Scripting (XCS) [57] attack to completely compromise an embedded device only using XSS and CSRF vulnerabilities. Other vulnerabilities are not as severe as the ones we just discussed and can be found in Table 5.6. Overall, we found vulnerabilities in 24% of the firmware images tested, which demonstrates the viability of our approach.

5.5.4 Presence of HTTPS

We also explored how often embedded devices web interfaces have HTTPS support. In our dataset 363 out of 1925 original firmware packages had at least one HTTPS certificate inside. We did not try to detect firmware instances which originally come without any HTTPS certificates and which actually generate the HTTPS certificates during a subsequent boot. This provides a lower bound estimate of firmware images that provide a web server with HTTPS support. Additionally, 60 firmware instances out of the 246 which started an HTTP web server, also started an HTTPS web server. We also expect this number to be lower than the reality as an HTTPS web server might not start for multiple reasons.

Table 5.6: Distribution of dynamic analysis vulnerabilities. NOTE: The count of vulnerabilities followed by "†" is not used elsewhere in this chapter when we mention a total number of vulnerabilities found. This is because they are known for very high false positive rates and low severity.

Vulnerability type	# of issues	# of affected FWs
Command execution	*51*	*21*
Cross-site scripting	*90*	*32*
CSRF	*84*	*37*
Sub-total HIGH impact	*225*	*45 (unique)*
Cookies w/o HttpOnly †	9	9
No X-Content-Type-Options †	2938	23
No X-Frame-Options †	2893	23
Backup files †	2	1
Application error info †	1	1
Sub-total low impact †	5843	23 (unique)
Total	**6068**	**58 (unique)**

Table 5.7: Distribution of vulnerabilities found by manual analysis (Section 5.3.5). NOTE: firmware images relate to similar products of one particular vendor.

Vulnerability type	# of affected FWs
Privilege escalation (full admin)	19
Unauthorized configuration download	19
Unencrypted configuration storage	19
Total HIGH impact	**19 (unique)**

5.5.5 Other Network Services

Embedded devices usually have a very specific purpose, for example to provide ADSL routing, to connect VoIP calls or to stream CCTV surveillance video. While our focus was on security of the web interfaces, which are often used for administrative purposes, we found interesting to report on the network services that are launched by those devices during the dynamic analysis. Indeed, those additional network services might be vulnerable on their own.

Examples of such services include TFTP [18], TR-069 [21], RTSP [20], Debug [19]. To gather such information, a straightforward method is to use the NMAP scanner [14] to scan the ports of the emulated firmware, however, this is not a very good option. On the one hand, performing an exhaustive TCP and UDP port scan was very slow, while on the other hand, the rapid scan option was missing non-standard network services and ports opened by emulated firmware packages. However, as our methodology uses a "white box" approach to

emulate a firmware (which allows us to see the emulated device from within) we can directly see the open sockets using the `netstat` tool inside the QEMU machine that emulates the device. Inevitably, the hosting system runs a minimal set of network services in order to be usable (e.g., sshd, rpcd). To get a list of network services started by the emulated firmware, we took one snapshot of the `netstat` output before we start emulating the device, and one snapshot after all the initialization scripts and web server launch completed. This approach provided us with a very clear and precise picture of the tuples—`[port-type, port-number, PID/program]`—which are opened by the emulated device. A detailed breakdown of network services is presented in Table 5.8.

Table 5.8: Distribution of network services opened by 207 firmware instances out of 488 successfully emulated ones. The last entry summarizes the 16 unusual port numbers opened by services such web servers.

Port type	Port number	Service name	# of FWs
TCP	554	RTSP	91
TCP	555	RTSP	84
TCP	23	Telnet	60
TCP	53	DNS	23
TCP	22	SSH	15
TCP	Others	Others	58
Total			**207 (unique)**

5.6 Discussion

5.6.1 Emulation Technique's Limitations

Although our approach is able to discover vulnerabilities in embedded web services that run inside an emulated environment, setting up this environment is sometimes difficult. In the following, we discuss several limitations we encountered and outline how they could be handled in the future.

Forced Emulation

Even though most of the firmware instances in our database are for Linux-based devices, they are quite heterogeneous and their binaries vary. Examples include `init` programs that have different set of command parameters or strictly requiring to run as PID 0, which is not the case in a chrooted environment. In theory, there should be a simple and uniform way to start the firmware, but this is not the case in practice as devices are very heterogeneous. In addition to this unless we have access to the bootloader of each individual device, there is no

reliable way to reproduce the boot sequence. Obtaining and reverse-engineering the boot-loaders themselves is not trivial because they are often not embedded in the firmware image. This usually requires access to the device, usage of physical memory dumping techniques, and manual reverse-engineering, which is outside the scope of this work. We emulate firmware images by forcefully invoking its default initialization scripts, (e.g., `/etc/init`, `/etc/rc`), however, sometimes, these scripts do not exist or fail to execute correctly leading to an incomplete system configuration. For instance, it may fail to mount the `/etc_ro` partition at the `/etc` mount point, and then, the web server is missing some required files (e.g., `/etc/passwd`).

Emulated Web Server Environment

Even if the basic emulation was successful, other problems with the emulated web server environment are common. For example, for many requests an emulated web interface can return the HTTP response codes `500 Internal Server Error` or `404 Not Found`. We manually inspected many of such cases to find the root cause. The HTTP error code 500 was mainly due to some scripts or binaries were either missing from the root filesystem or did not have proper permissions. The HTTP error code 404 was often due to the wrong web server configuration file being loaded. Loading a wrong configuration file made the document root to point to a wrong directory and hence the HTTP error code 404. To overcome this, we try to emulate the web interface of a firmware using all combinations of the configuration files and document roots we find in this firmware.

Imperfect Emulation

The ability to emulate embedded software in QEMU is incredibly valuable, but comes at a price. One big drawback is that some very common peripheral devices are missing in the emulated environments. One of the most common emulation failure is related to the lack of non volatile memories (e.g., NVRAM). Such memories are used by embedded devices to store boot and configuration information. The drawback is confirmed by researchers emulating individual firmware images [84, 118].

There are several approaches to overcome such limitations. One is to have an universal or on-the-fly NVRAM emulator plugged into the QEMU hypervisor. For example, it can be instrumented at kernel-level or implemented using `Avatar` [203]. Another approach is to intercept calls to the commonly used `libnvram` functions (such as `nvram_get` and `nvram_set`) and return fake data. This technique is described in detail elsewhere [84, 118] and we will plan to integrate it in our future work. However, we expect that in many cases this will still be problematic. The NVRAM is used as non volatile storage, e.g., for settings. Providing a fake

NVRAM device with incorrect data will not solve all problems and probably will introduce new ones.

5.7 Future Work

Our future work will consist in improvements to our framework. First, additional research is required to improve emulation quality. Second, the number of vulnerabilities found (in particular by static analysis) is large and therefore, non trivial to manually verify. Verifying them would require to automatically synthesize web exploits, which is not currently possible. Finally, responsibly disclosing vulnerabilities is time consuming and difficult (and in our experience is worse with vendors of SOHO devices). It becomes an open challenge when it needs to be performed at a large scale.

5.8 Summary

In this chapter, we presented a new methodology to perform large scale security analysis of web interfaces within embedded devices. For this purpose, we designed a framework leveraging off-the-shelf static and dynamic analysis tools. Because of the limitations in static analysis tools, we created a mechanism for automatic emulation of firmware images. While perfectly emulating unknown hardware will probably remain an open problem, we were able to emulate systems well enough to test the web interfaces of 246 firmware images. Our framework found serious vulnerabilities in at least 24% of the web interfaces we were able to emulate. When including the static analysis phase, 9290 issues were found in a total of 185 firmware images. This includes 225 $high\ impact$ vulnerabilities that we were able to verify.

Finally, our experiments and results confirm that the security of many of those devices is seriously lacking. We therefore aim at running this framework as a continuous process. This can help to improve its quality and to keep on finding vulnerabilities in such devices, hoping they will be addressed by the vendors. Such a service could also be useful to device vendors who can benefit from automated security testing before shipping their products. We hope our system can help make the Internet and IoT more secure.

Chapter 6

Scalable Firmware Classification and Identification of Embedded Devices

6.1 Introduction

A firmware image is in general custom made for a specific device and a device model is running a particular firmware file. This is relatively easy for a human to follow during manual analysis (e.g., firmware upgrade process). However, because devices are so diverse, it is not trivial for computers and automated systems to link a device model and a firmware image.

For instance, when manually downloading a single firmware file from a vendor site, it is often relatively easy for a human to know the vendor and the device for which the firmware is intended. However, for an automated system which crawls thousands of firmware files from unstructured download sites it is not a trivial task to categorize firmware files by device class or even by vendor. We identified and described this problem as the "Firmware Identification" challenge in Chapter 4.

6.1.1 Open Problems

Within this context we identify and formulate two problems as follows: First, how to *automatically* and *accurately* label the brand and the model of the device for which the firmware is intended. Second, how to *automatically* identify the vendor, the model and the firmware version of an arbitrary remote online device.

Those steps need to be preformed in a reliable way which is independent of the device, the vendor, or the custom protocols running on the device.

6.1.2 Overview of our Approach

In our method, we apply Machine Learning (ML) to classify firmware files according to their vendor or device type. We use two widely adopted algorithms, Random Forests (RF) and Decision Trees (DT), based on their implementation in `scikit-learn` package [168]. We explore several feature sets derived from the characteristics of firmware images, such as file size, file entropy and common strings. Then, we recommend the optimal feature set for this type of classification problems and show that our approach achieves high accuracy. Moreover, using sound statistical methods such as confidence intervals we estimate the performance of our classifiers for large scale real world datasets.

We then perform web interface fingerprinting both on previously classified firmware files, which we then emulate, and on real devices. We build a fingerprinting database based on these emulated and real devices. Then we can match an unknown embedded web interface to the list of known web fingerprints in our database by using multiple matching metrics, such as the sitemap or the HTTP protocol Finite-State Machine (FSM). Finally, we use multiple scoring systems to rank the fingerprint matches.

6.1.3 Contributions

In summary, we make the following main contributions:

- We apply machine learning in the context of firmware classification, and we study and propose the firmware features that makes this possible.
- We show that using machine learning it is possible to automatically classify sets of firmware images with high accuracy.
- We study the fingerprinting and identification of embedded devices and their firmware version by performing multi-metric web interface fingerprinting of physical and emulated embedded devices.

6.2 Firmware Classification and Identification

In this section we present our classifier. We use supervised Machine Learning (ML), i.e., the ML algorithm must be trained with a set of (manually) annotated samples before it can classify unknown samples. We experimented with *Decision*

Trees (DT) and *Random Forests (RF)*. Compared to *Support Vector Machines (SVM)*, DT and RF algorithms do not require cross-validation, are able to better handle non-linear features, and are easier and faster to train.[1]

Supervised ML algorithms require *data features* that they can use to partition and distinguish the learned classes of data. This means that feature selection is an important step towards how successful the learning and classification will be. Feature selection is usually specific to the domain to which the ML is applied, and this selection must be carefully performed and evaluated.

We therefore first present our dataset, then the features we selected and finally we measure the performance of the classifier.

6.2.1 Dataset

We first obtained access to a large firmware dataset, which contains more than 172K firmware images as detailed in Chapter 4. From this initial dataset, we selected 215 firmware images from 13 vendors. Although we did not perform a truly random selection, we tried to ensure that the set of selected firmware images contains enough variance in terms of vendors, product types, and firmware image properties (e.g., size, file format). Those vendors manufacture several type of devices: routers, home automation, multi-function SD cards, Set Top Boxes (STB), VoIP devices, avionics radars, Closed-Circuit TeleVision (CCTV) and Digital Video Recording (DVR) systems. We will refer to these 13 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 strategies are very diverse. Each classification category contains between 5 and 54 firmware images, with an average of 16 images per vendor (i.e., category).

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.

6.2.2 Features for Machine Learning

A classification of a firmware file can be performed at vendor level or at device line level, depending on the granularity objectives. For consistency, we will refer to vendor categories or device line categories simply as *classification categories*.

Firmware file size. The file size of a full firmware upgrade for an embedded device is directly linked to the hardware design and the functionalities of the

[1] https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm#ooberr

device. In other words, the size of the firmware is dictated by the device's memory constraints and the minimal set of binaries required to provide the functionality. At the same time, a firmware upgrade file cannot exceed the limited memory available in the particular device lines which it targets. Hence, this motivates using *firmware file size* as a good feature to discriminate between firmware images for devices from different classification categories.

Firmware file content entropy. Most of the vendors use their own procedures to build and package a firmware upgrade. Vendors then wrap them into non standard file formats suitable for the device line. This makes the firmware images from a particular classification category to have specific distribution and density of the information they contain. Hence, we propose to use information theory metrics as features for ML. In this sense, we use the following characteristics[2] of the firmware files as ML features:

- File entropy, i.e., the informational density of bits per byte
- Arithmetic mean of file bytes
- File compressibility percentage, i.e., an empirical value that is an upper bound of the Kolmogorov complexity
- Serial correlation value
- Monte-Carlo value and its estimation error
- 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, contains strings. These strings may contain copyright, debugging or other information. They often contain vendor or device specific information. Therefore, the strings in a given firmware file represent a pretty good fingerprint of the corresponding firmware, device and vendor. As a consequence, the intersection of strings of each firmware file within a particular classification category may represent a strong classification feature. In other words, suppose an unknown firmware sample contains a string that is found within strings intersection of a *classification category* A. Then there are high chances that the firmware file is somehow related to the files in the *classification category* A.

At the same time though, many firmware files (spread across different classification categories) may contain strings that are common across multiple classification categories. For example, this happens if the firmware uses common Free Open Source Software (FOSS) code such as Linux kernel or OpenSSL libraries. In

[2]We extract these characteristics from the output of the `ent` Linux utility.

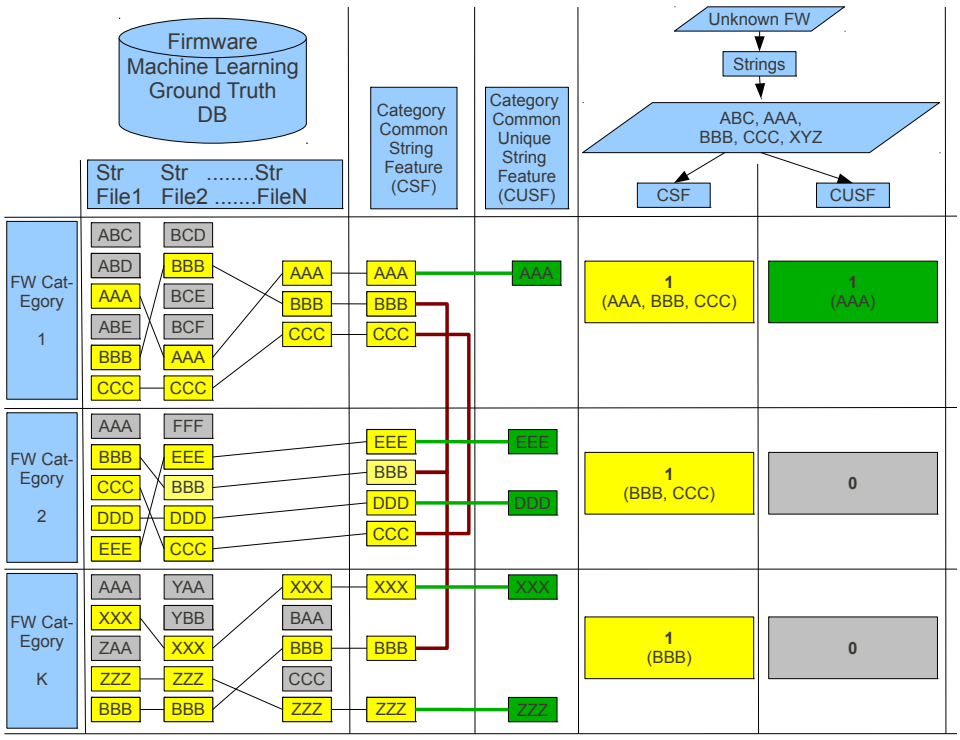

Figure 6.1: Derivation and assignment of strings-based features.

this case, if using the "naive" string-to-category matching, an unknown firmware 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 and are not contained in any classification 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)*. A trained or unknown file gets the CSF and CUSF feature value (i.e., 0 or 1) assigned as depicted in Figure 6.1.

Fuzzy hashes. Fuzzy hashing is the technique to compute the percentage of similarity between two different files. The cryptographic hashing is used to determine if two different files are exact equals, while the fuzzy hashing is used to determine if two different files are homologous, i.e., are similar but not exact equals.

Intuitively, firmware files from a given classification category should generally be "fuzzy hash similar" among themselves then cross-category. For this reason, 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 (default similarity threshold is 50%), the fuzzy hash feature of that category is set to 1. Otherwise, if none of the fuzzy hashes from a category's fuzzy hash list does match above the threshold, the fuzzy hash feature of the category is set to 0. Therefore, each classification category in the training set adds one additional feature.

Surprisingly, including the fuzzy hash similarities as features proved to result in worse classification accuracy as discussed in Section 6.2.4 and Section 6.2.5.

6.2.3 Experimental Setup

Running supervised machine learning experiments requires training sets. Since our dataset have the classification categories of varying lengths (Section 6.2.1), 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 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, running the training and classification, and finally computing the average classification error. For any experiment run, we use both Decision Trees and Random Forests algorithms so that we can compare their performance under various conditions.

Listing 6.1: Pseudo-code for preparing and running the ML experiments.

```
for P in [10% to 90% step 10%]
begin
  for R in [1 to 100 step 1]
  begin
    RF_Exper_Perc_P_Run_R(ClsfyTrue, ClsfyFalse) =
      ML_RandomForest_Classify(
        set_train = random_sampling(dataset, P),
        set_classify = sets_diff(dataset, set_train),
        set_features
      )
    DT_Exper_Perc_P_Run_R(ClsfyTrue, ClsfyFalse) =
      ML_DecisionTree_Classify(
        set_train = random_sampling(dataset, P),
        set_classify = sets_diff(dataset, set_train),
        set_features
      )
  end
  RF_Exper_Perc_P_Avg_Error =
    Avg( RF_Exper_Perc_P[ClsfyFalse] )
  DT_Exper_Perc_P_Avg_Error =
    Avg( DT_Exper_Perc_P[ClsfyFalse] )
```

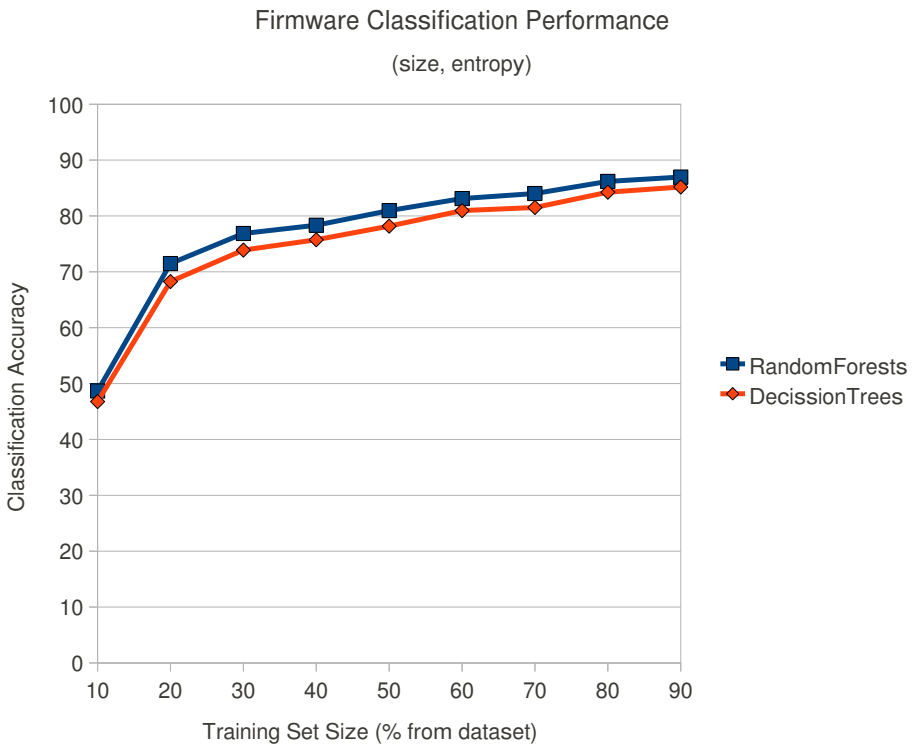

Figure 6.2: Firmware classification performance using [size, entropy] feature set of the firmware files.

end

6.2.4 Evaluation

The performance of the firmware classification for various machine learning algorithms, feature sets and size of the training sets (i.e., percentage from the original dataset) is summarized in the Figures 6.2, 6.3, 6.4, and 6.5.

Firstly, according to the intuitive expectation, we can observe that the classification accuracy improves with the increased size of the training set. Secondly, contrary to the intuitive expectation, the addition of the fuzzy hash similarity features reduced the accuracy. We expected that the fuzzy hash similarity features would increase the accuracy of the classification. However, this addition made both the RF and the DT classifiers perform worse. With these features 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]`. At the same time, the RF classifier in this setup failed to

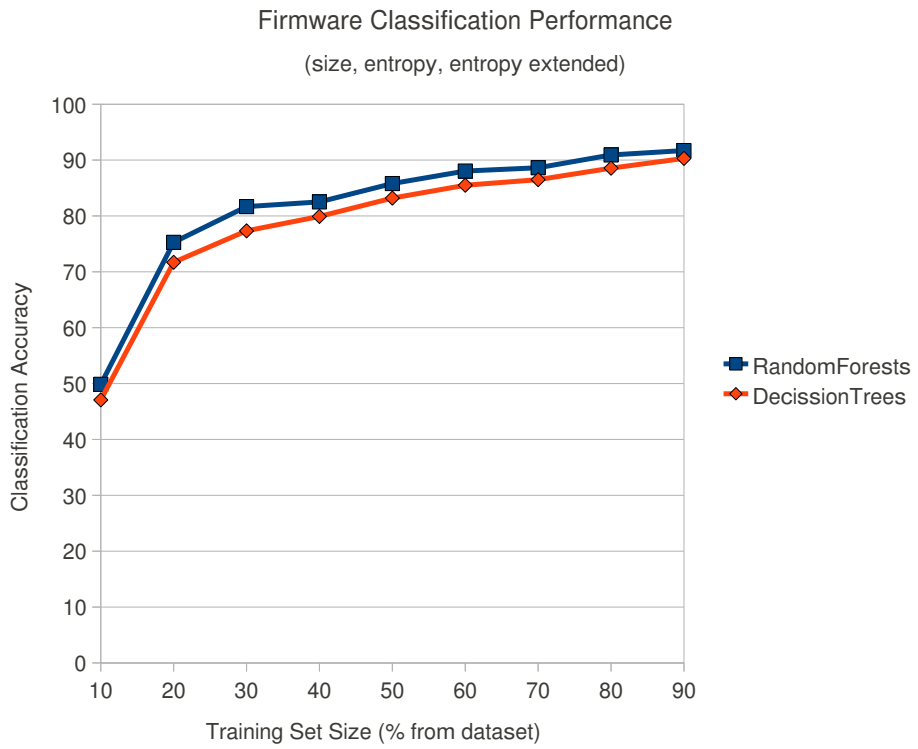

Figure 6.3: Firmware classification performance using [size, entropy, entropy extended] feature set of the firmware files.

perform at least marginally better than the RF classifiers based on basic feature sets mentioned above. One explanation for this bad performance could be the fact that 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 above observations, we conclude that the feature set `[size, entropy, entropy extended, category strings, category unique strings]` constitutes the best choice. It provides best accuracy when used with the RF classifiers. Using this feature set, the RF classifier achieves more than 90% classification accuracy when the training set is based on at least 40% of the known firmware files. We argue that such a percentage to form a training set is feasible in real-life. In other words, let us consider the case when a device is expected to have two firmware versions during its life-time. For example, the original firmware running on the device and the firmware update to fix a critical bug. Having access to the original firmware (e.g., downloading from the vendor's device-recovery site or using techniques similar to flash dumping) means one can already have access to 50% of that device firmware set.

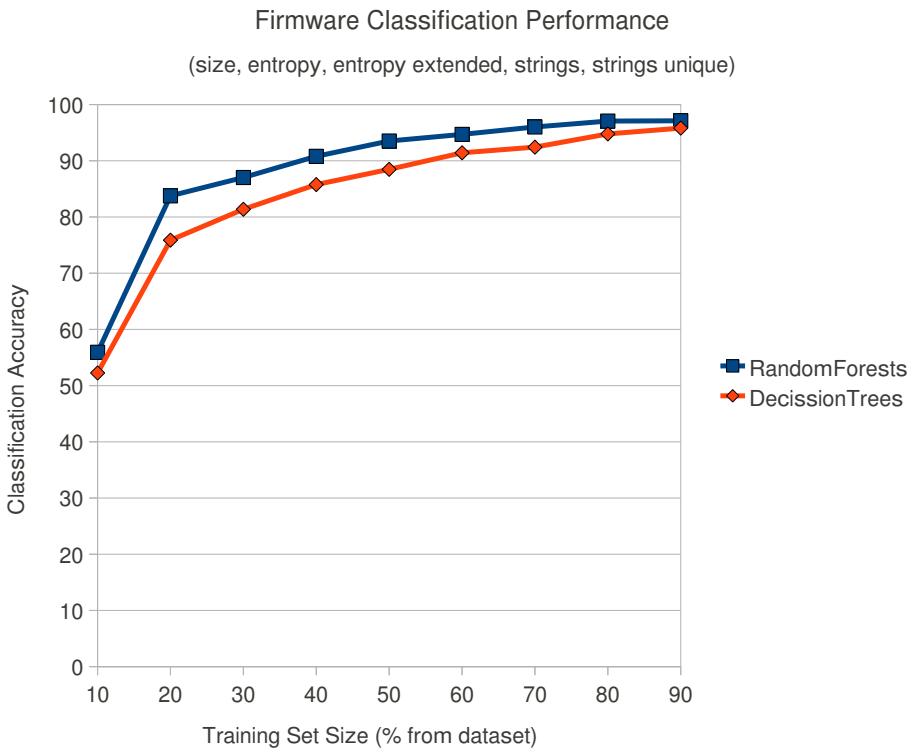

Figure 6.4: Firmware classification performance using [size, entropy, entropy extended, strings, strings unique] feature set of the firmware files.

Therefore as a reference we can safely use the training set size of 50% of known firmware files. Under these assumptions, the RF classifier achieves a very high accuracy of 93.5%, while the DT classifier classifies correctly only 88.4% of 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.

While we make all the effort to identify the most reliable feature set and learning classifier, the *generalization of learning* is a known open problem in the machine learning field [53, 56, 92].

The current machine learning algorithms performances cannot be guaranteed on another dataset (e.g., bigger). We try to compensate this limitation with statistical methods, such as confidence intervals. In this context, we used statistical confidence intervals [61] to evaluate the accuracy of our technique when applied to real-world populations of firmware images.

For example, let us consider any firmware in the original dataset of 172K firmware

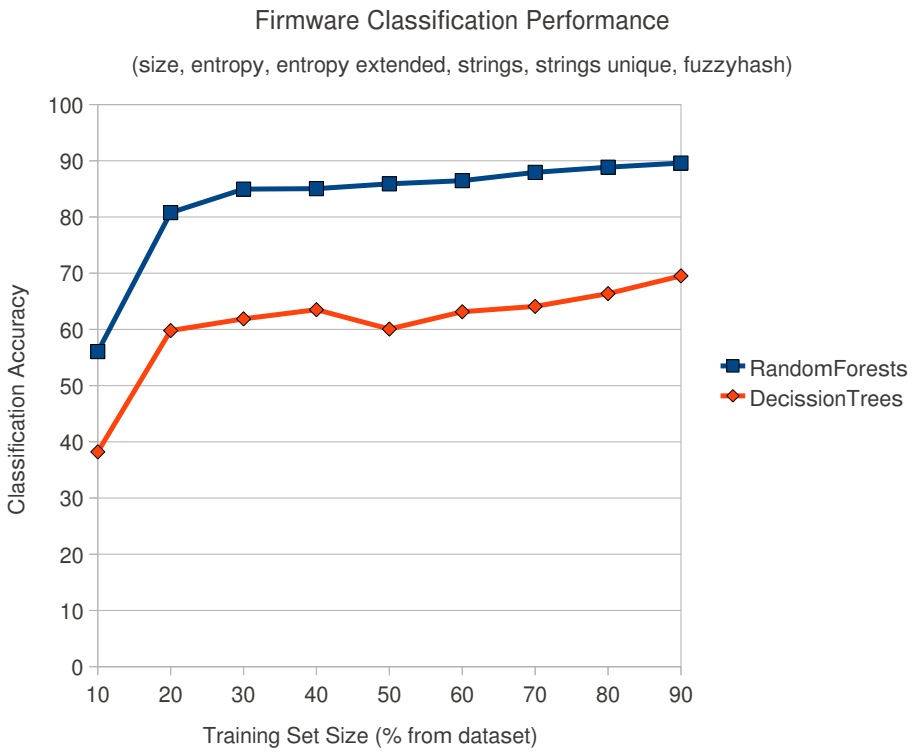

Figure 6.5: Firmware classification performance using [size, entropy, entropy extended, strings, strings unique, fuzzy hash] feature set of the firmware files.

images. 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 Random Forest 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 or by using techniques similar to Google's CAPTCHA for recognizing OCR text or for mapping street numbers from street views. We leave this interesting challenge for future work.

6.2.5 Discussion

Some vendors have dozens of product types and therefore have many different firmware branches. In such cases, to achieve finer granularity of firmware classi-

fication, the training and the *classification categories* can be formed per type of device, the training dataset has to be split accordingly.

There are other cases when a firmware upgrade is wrapped into multiple layers of packaging. This makes the firmware more challenging to unpack and it is harder to understand at which layer exactly the firmware upgrade file starts or ends. For example, a firmware image (e.g., `.bin, .img, .fw`) is provided in compressed formats (e.g., `.zip, .gz`). Classifying compressed formats using our features set (Section 6.2.2) is not trivial, since the entropy of compressed files is very similar regardless of its content. These challenges can be solved as follows. Any firmware package that must to be classified is also unpacked using best-effort or brute force unpacking. The unpacked tree is traversed one depth-level at a time. All the files at a particular unpack level are classified using exactly the same optimal approach described above. Once a successful classification is achieved, the unpack sub-tree under the freshly classified file is considered to belong to the particular vendor and device line. This process can stop or continue for other unclassified files in the unpack tree, depending on the goal and constraints.

Finally, more experimentation with the the fuzzy hash similarity feature set is required to take full advantage of the fuzzy hash information. We plan to address this in future work as follows. Instead of using a hard threshold of 50% and a feature value of 0 or 1, we can compute a fuzzy has similarity feature value between 0 and 100 as follows. The fuzzy hash of an unknown firmware is compared to all fuzzy hashes in a given classification category, and the comparison values are used later. Then the *arithmetic mean* is computed for all the comparison values for the given category. This will result in a value between 0 and 100. In practice, these steps are repeated for all the classification categories to compute the fuzzy hash feature value for the unknown firmware, corresponding to each category.

6.3 Device Fingerprinting and Identification

Many approaches exists for fingerprinting and identification of computing device and sensors [60, 90, 113, 144, 163, 176]. 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 that can be emulated or can provide additional information for a reliable device identification.

At the same time, it is known that the embedded devices lack the user interfaces of desktop computers, such as keyboard, video, mouse. However these devices need to be administered one way or another. 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. For this reasons, 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 or can prove helpful in order for the fingerprinting to reliably work regardless the way the devices operate.

Therefore, in contrast to existing work we propose an approach that fingerprints the devices at a high level possible, the *embedded web interface* level. This approach can also take advantage of the firmware contents and the device emulation based on the firmware images alone. Previous work touched some aspects of our fingerprinting techniques. However, those either suggest manual approaches [162] or do not provide enough insights and evaluations [183].

6.3.1 Dataset

In our fingerprinting experiments we used 27 firmware images originating from 3 vendors that split across 7 functional categories. Out of these 27 emulated firmware images, 9 of them where also part of the firmware ML classification experiments. Specifically these 9 firmware packages were classified by our ML firmware model with an accuracy of 100% using Random Forest (and around 99.5% using Decision Tree). Additionally, we used 4 physical devices from 2 vendors that cover 4 functional categories.

The detailed breakdown of the 27 emulated devices is as follows:

- *Network Video Servers* – Brickcom VS-01Ae, firmware version v3.1.0.4
- *IP Cameras* – Brickcom CB-100Ae, CB-100Ap, CB-102Ae_V2, CB-102Ap_V2, FB-100Ap, FB-100Ap_V2, FB-130Np_V2, OB-100Ap, altogether running ten firmware images
- *PTZ Dome Cameras* – Brickcom FD-100Ae, FD-100Ae_V2, FD-100Ap, FD-130Ae_V2, FD-130Ap_V2, FD-130Np_V2, MD-100Ap_V2, VD-100Ae_V2, VD-130Ap_V2, VD-300Ap_V2, altogether running ten firmware images
- *Multi-function SDcards* – Transcend WiFiSD, firmware versions v1.2.1 and v1.6
- *Wireless Routers* – NetGear WNR612, firmware version v3-v1.0.0.11
- *DSL Modem Routers* – NetGear DG632, firmware versions v3.3.0.a.cx and v3.40
- *Wireless Access Points* – NetGear WNAP210, firmware version v2.0.27

The detailed breakdown of the 4 physical devices is as follows:

- *Wireless Routers* – DLink DIR-632, firmware version DD-WRT v24-sp2 (01/24/13) std (SVN revision 20548)
- *Ethernet Routers* – DLink DI-604 (F4), firmware version v3.14
- *DSL Modem Routers* – DLink DSL-2500U (A1), firmware version v3-06-04-0Z00-RU
- *Wireless DSL Modem Routers* – Netgear DG-834GT, firmware version v1.03.23

6.3.2 Metrics for Fingerprinting

We propose six different metrics (i.e., features) that are computed for each training or unknown embedded web interface. These metrics are: `sitemap`, `FSM`, `fuzzyhash header`, `fuzzyhash content`, `cryptohash header`, `cryptohash content`. We present them below and motivate the choice.

Sitemap. A sitemap is a list of pages of a website which are publicly or privately accessible. Most of the times, each website (or a web application) has its unique structure of sitemap. That is, files and URLs that exist in one website do not necessarily exist in another one, even if they run on the same web server. Therefore, a sitemap of a web application can be used one of its uniqueness feature.

To this end, we leverage this fact and create a fingerprint based on this assumption. More precisely, if we want to categorize the web interface of an unknown embedded device, we can try to 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 perfectly matches with a known one in our database, we can classify it with high confidence as belonging to an embedded device running a specific firmware version.

HTTP Finite-State Machine. The *HyperText Transfer Protocol* (HTTP), as defined in RFC 2616 [111], is a stateless application-level protocol for distributed, collaborative, and hypermedia information systems. The protocol is defined as a conversation between the client and the server, where text messages are transmitted in an alternating way. Messages consist of *requests* from client to server and *responses* from server to client. Both types of messages contain a *start line*, zero or more *header fields* (known as *headers*), an empty line that indicates the end of the headers, and optionally a *message body*. Each

line ends with a line-terminator, denoted as `CRLF`. Each header consists of a case-insensitive name followed by a colon and the field value.

For our fingerprinting and detection purposes we focus only on the server responses, i.e., the responses from the embedded web interfaces. 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. In this work, we leverage these differences to identify the type of server involved in a specific HTTP conversation.

More precisely, 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. To this end, in order to represent the headers' order of an HTTP response, we define a $header\ sequence$ as a vector $\overrightarrow{H} = (h_1, h_2, ..., h_n)$ with header names as elements. For each HTTP response of a known web server S, we create and store a pair $(\overrightarrow{H}, S)$ of the request's header chain $\overrightarrow{H}$ and the server's name S. To classify an unknown server, we form its sequence $U_{\overrightarrow{H}}$ and compare all our labeled $\overrightarrow{H}$ with the unlabeled sequence $U_{\overrightarrow{H}}$. If we find that a sequence $\overrightarrow{H}$ is equal to the observed sequence $U_{\overrightarrow{H}}$, we assume that the corresponding server S generated this response. In essence, we have implemented an HTTP Finite-State Machine (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 content and headers. We expand the FSM approach by using not just the header names of an HTTP response but also their actual values. We investigate when and how these values can be used to classify an unknown web interface. Although some headers will always display the same information (e.g., the header $Server$ shows information about the web server), few other headers will not remain constant over time (e.g., the header $Date$ represents the date and time at which a message was originated). 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 (e.g., `favicon.ico`) will results 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 $Date$ (that always displays different values even for a static web resource) do not affect our metrics and matching. Note that we do not generate regular expressions for the body of the response message.

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 contains the hashed message body. If we have a perfect match between an unknown HTTP response and an HTTP response contained in our fingerprints database, we can successfully fingerprint the device that sent this response. It is worth mentioning that many times a 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 narrow down the number of possible devices that match this response. Overall, a successful fingerprinting contains a combination of all the metrics we apply.

These two metrics are a better fit for resources that are statically delivered to the client (e.g., user's web browser), such as JavaScript, CSS, and image resources.

Fuzzy hashing of content and 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 mostly happens because even a small modification on just one byte in a large byte stream can cause the cryptographic hash function to generate a completely different hash value. For example, consider a configuration page which allows the users to set custom SSIDs for a wireless router or custom host names for an Ethernet router. Even though the server-side page is the same, the page returned to a client or crawler will have two different cryptographic hash values for two different SSIDs or host names. To counter this behavior we use fuzzy hashing which we introduced in Chapter 2. We chose to use Context Triggered Piecewise Hashes (CTPH) [147] to define the similarity between an unknown HTTP response and a list of HTTP responses for which we know their fuzzy hashes. 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.[3] If the similarity between an unknown and a known HTTP response exceeds an empirically calculated threshold, we can successfully classify this unknown device.

These two metrics are a better fit for resources that are dynamically generated or are interpreted on the server-side, such as CGI and PHP resources.

6.3.3 Scoring Systems for Metrics

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

Metrics majority voting score. In this scoring system, each metric of each fingerprint match is ranked in decreasing order. The fingerprint match that ranks highest on most of its metrics is considered to be the most accurate match to the unknown sample.

[3] http://ssdeep.sourceforge.net/

Uniform and non-uniform weighting scores. In these scoring system, each metric value of a fingerprint is assigned a weight. Then, for each metric 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 fingerprint match whose total value ranks highest is considered to be the most accurate match to the unknown sample.

For our evaluation, we used the uniform weights of 16.6% for each of the six metrics. We also used the empirically found weights for each of the six metrics: 4% for `sitemap`, 4% for `FSM`, 1% for `fuzzy hash header`, 1% for `fuzzy hash content`, 10% for `crypto hash header`, 80% for `crypto hash 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 [161] and sensors data [143]. 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.

6.3.4 Experimental Setup

We start by emulating the 27 firmware images and connecting up the 4 physical devices. At this step we apply the firmware emulation technique previously introduced in Chapter 5. 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 make sure that the list of IP addresses is randomized every time it is created. 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 metrics 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 of successful and erroneous identification rates based on results from each experimental run.

6.3.5 Evaluation

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. The fingerprinted web interfaces contained around a third 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 6.2.4, 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 6.3.3. 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 can be explained by the high weights assigned to non-accurate and noisy fuzzy hash metrics.

6.3.6 Discussion

Embedded web interface's URLs that majorly affect the device functionality, e.g., `reboot.php`, `FW-upgrade.cgi`, represent a particular challenge. Accessing such URLs during fingerprinting or matching may cause the device to go offline or malfunction right in the middle of the identification process. As a result the fingerprint or match will be partial and may produce wrong results. Undoubtedly, automatically detecting such URLs and excluding them from the identification process may help prevent such behavior. One way to achieve this detection is by doing manual annotations on a case-by-case basis, but this obviously does not scale. Other ways to achieve this is by using static and dynamic analysis. Alternatively, it is possible to perform a first sequential run on each URL of an embedded web interface and check the "liveness" of the embedded web interface after each URL is accessed. Even though this method is not 100% accurate since the device can go offline or malfunction for other reasons as well, it may help removing the perturbing URLs and establish a "safe" list of URLs to be used for device identification. Finally, though removing few URLs from the identification process might slightly reduce the exploratory surface of the device's web interface, the number of such URLs in practice is extremely low compared to all other resources in the embedded web interface (e.g., images, JavaScript,

styles, static help files). Thus the impact of their removal on the the scoring is expected to be minimal.

Additionally, we designed our fingerprinting tools in a robust manner so that they support HTTP authentication and HTTP cookies. This functionality is required since many embedded devices user either HTTP basic authentication or HTML forms authentication which is stored in session cookies. Having this functionality enables us to create more granular fingerprints and, depending on the embedded web interface, identify if a given access to a connected device is `unauthenticated, unauthenticated as admin, ... , unauthenticated as userN`.

Finally, our embedded web interface identification can be further improved by combining our fingerprint data with results from complementary fingerprint levels, such as device [144], CPU and clock [144], OS [190], network stack (e.g., NMAP) [114], combined (e.g., Nessus [4]).

6.4 Usage Scenarios

While taking a research-oriented approach to the open problems formulated in Section 6.1.1, with this work we also aim at providing practical results and usability. Thus we consider that providing real-life examples and applications are equally important. We do so by providing few usage examples for the techniques we proposed.

6.4.1 Device Fingerprinting and Identification

Defensive **use of the technique.** In some cases, our device fingerprinting and identification technique may be used to scan a home, SOHO or enterprise network, and fingerprint the detected embedded web interfaces. Subsequently, the fingerprint information may be used to identify the device vendor, the device model and its firmware version. Additionally, this information can 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. Finally, these new user annotated fingerprints can be added to our database in an anonymized manner and can help increase the accuracy of our platform.

Offensive **use of the technique.** In other cases, a penetration tester may be performing a black-box penetration testing. She may use our device fingerprinting

[4] http://www.tenable.com/products/nessus-vulnerability-scanner

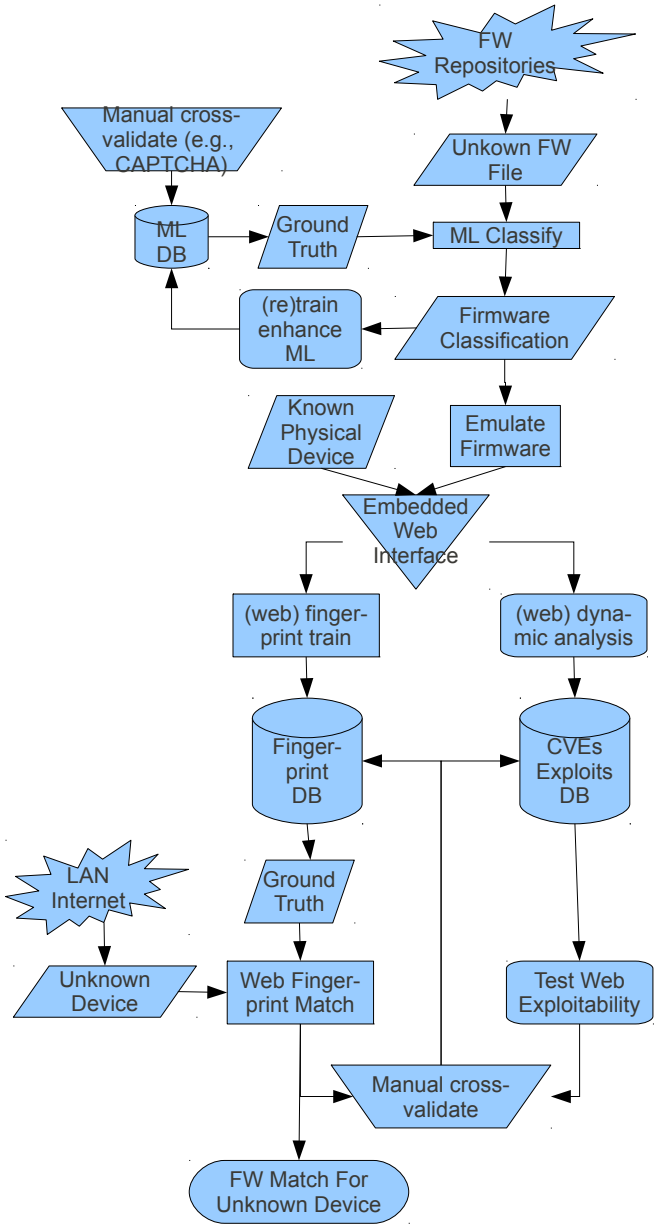

Figure 6.6: End-to-end process where our firmware classification and device identification techniques are applied.

and identification technique to identify the exact device model and the firmware version of an unknown embedded device encountered in the network under test. With this information, CVEs or exploits could be retrieved for the particular device model and firmware version. This may help increase the test's success rate and decrease the time required to perform the test.

6.4.2 Firmware Classification

Correct identification and classification of a firmware could be extremely beneficial. For example, once a firmware file is correctly classified according to its vendor, category or model, various optimization could be applied. First, only a specific set of firmware unpackers are run on the firmware, skipping the brute force unpacking, thus saving processing time and providing the results faster. Second, once the vendor is known as a result of a successful classification, then very specific static analysis techniques and tools may be applied according to the knowledge about vendor's development practices.

6.4.3 Towards Fully Automated System – "Crawl. Learn. Classify. Identify. Pwn."

It is possible to achieve a fully automated system depicted in Figure 6.6 by combining the two techniques we propose. First, crawlers or web-submission portals collect firmware images, which are then automatically classified and identified. Once this is done, an emulator for the embedded web interface can be automatically started [41]. The web interface emulator may allow finding new vulnerabilities via dynamic analysis and provides the opportunity to fingerprint it. This creates a growing fingerprint database that is automatically improved. Second, Internet and private network scanners or web-submission portals collect the IPs of presumably embedded devices. Those IPs are analyzed and matched against the fingerprint database. Finally, once the match contains an obsolete firmware version, the system could automatically upgrade the firmware to the latest more secure version or it could notify the owner of the inadvertently exposed online device. The system could also automatically execute a list of penetration tests or exploits related to the vulnerable firmware version running on the device.

6.5 Summary

In this chapter we presented two complementary techniques, namely *embedded firmware trained classification*, and *embedded web interface fingerprinted identification*. We proposed machine learning for the firmware classification challenge and explored multi-metric score fusion to address the web interface identification problem. With high confidence for real-world large scale datasets, our

tests demonstrate that the classifiers and features we propose can achieve 93.5% accuracy in firmware classification and 89.4% accuracy in device identification.

Chapter 7

Conclusions

This dissertation presented novel techniques for automated large scale analysis of software security in embedded devices and their firmware. We implemented these techniques into a fully working framework and validated its effectiveness against real-world data (i.e., firmware images and online devices).

We started the journey of this thesis with a simple yet effective framework. Framework's crawling modules efficiently collected firmware images for the purpose of subsequent analysis. We used simple crawlers based on a mix of customized site scrapers, custom search queries and support download locations. We were able to collect 172K potential firmware images. Our further estimates showed that with a confidence of 95% there should be at least $34\% \pm 8\%$ real firmware images in this dataset. This dataset then required unpacking and security analysis. The unpacking module of our framework is based on a customized extension of the Binary Analysis Toolkit (`BAT`) platform, but can be easily extended in the future with other unpacking frameworks such as `binwalk`. Even though the unpacking module cannot yet guarantee complete unpacking of any firmware, it unpacked around 75% of processed firmware candidates. Ultimately, simple static analysis modules allowed us to find 38 new vulnerabilities in 693 firmware images. For example, such modules analyze weak passwords in `/etc/passwd`, collect and track online SSL and SSH private keys, and check for simple security misconfigurations and obvious backdoors. These experiments allowed us to identify five important challenges associated with the security analysis of embedded firmware at large scale. Moreover, some challenges were strongly linked to insufficiently researched areas that we successfully explored afterward.

We then improved our system by introducing static and dynamic automated analysis of embedded web interfaces at large scale. The approach is based on the distributed and architecture-independent emulation of the root filesystems extracted during firmware unpacking. We also developed and employed a set of automated heuristics to increase the success of both the firmware emulation and the embedded web interface launch. Once the embedded web interfaces are

launched, our framework applies state of the art static and dynamic analysis tools on them. Our fully automated system discovered in a matter of only few hours high-impact vulnerabilities (e.g., command injection, XSS) in at least 20% of emulated embedded web interfaces. At the same time, it could automatically emulate the embedded web interfaces within 15% of web-enabled firmware packages. Increasing these success rates in an automated and intelligent manner is a challenge that we want to address in our future work. Our approach is flexible which means that new emulation and analysis techniques can be easily added in the future. Moreover, though it is now known that many embedded devices are insecure, our system is really the first demonstrating the possibility to fully and feasibly automate dynamic analysis of heterogeneous embedded firmware at scale.

Finally, we enhanced our system with additional intelligence by employing Machine Learning (ML) and classification techniques. To classify collected firmware files, we explored Random Forests (RF) and Decision Trees (DT) algorithms in combination with several feature sets. On our firmware dataset, we showed that the RF algorithm with the feature set of `[size, entropy, entropy extended, category strings, category unique strings]` is the best choice among the four main feature sets we explored. For example, our system achieved more than 90% classification accuracy when the training sets were based on at least 40% of each known firmware category. To classify online embedded devices, we explored web interface level fingerprinting based on multi-metric score fusion techniques. Our system relies on fingerprints of the embedded web interfaces computed over six metrics. Then it ranks the fingerprint metrics using three scoring systems, and uses score fusion technique in the final evaluation of the best fingerprint match. We also reasonably motivated our choices for the metrics and the scoring systems in the context of embedded web interfaces. For example, on average our system achieved 89.4% accuracy in device identification based on a database of 31 fingerprints of embedded web interfaces. Ultimately, we demonstrated that it is possible to classify firmware files and identify online embedded devices with high accuracy.

7.1 Future Work

Future work will focus on building clean, annotated and representative datasets of firmware images and device emulations. For example, these datasets could be used by practitioners as "ground truth" in future experiments. This would allow to evaluate the effectiveness and the efficiency of novel techniques aimed at discovering vulnerabilities in embedded devices and their firmware. Furthermore, this would allow a fair and reasonable comparison between different techniques and approaches.

Another improvement planned as future work is to leverage a CAPTCHA-like

mechanism to build in an incremental manner a clean training set for firmware classification. For example, unclassified firmware files could be randomly presented for classification to multiple users of our http://firmware.re service. For cross-validation of the users' response, a firmware with a known good label is also presented as a challenge to the users along with the unclassified ones. One challenge in this process could be the design of a visually compelling representation of the firmware files presented to the users. Finally, once an unclassified firmware file achieves a classification threshold in a particular category, it is added to the training set under the category's label.

We also plan to develop novel tools and techniques for static analysis, with a particular focus on their applications to securing the code within firmware images. For example, these could be static analysis tools for web technologies that are not well covered by the state of the art, such as Lua or Haserl. In other instances, these could be binary static analysis methods for the myriad of less common CPU architectures found in the embedded and IoT devices.

Finally, we plan to develop, deploy and monitor robust and maximally realistic honeypots emulating heterogeneous embedded devices. This would allow to capture and analyze at early stages novel threats, exploits and malware that target a diverse range of embedded and IoT devices.

Appendix A

Résumé de la thèse en français

A.1 Introduction

Les systèmes embarqués sont omniprésents dans notre vie quotidienne et ils sont de plus en plus présente dans de nombreux environnements informatique et en réseau. En fait, plusieurs rapports estiment une augmentation du nombre de dispositifs embarqués dans les prochaines années [133, 170]. Cisco prédit qu'il y aura 50 milliards de *dispositifs embarqués connectés* en 2020 [72]. Ces appareils seront produits par de nombreux fabricants différents et seront présents dans de nombreux modèles différents. Chacun aura probablement plusieurs versions de firmware, conduisant à un grand nombre de versions de firmware. Comme indiqué dans le Chapitre 4, des centaines de milliers d'images de firmware sont déjà disponibles, qui est juste une estimation de la limite inférieure de firmware publiquement observables. Le nombre de fichiers du firmware probablement seulement va croître avec le nombre de nouveaux dispositifs embarqués développés et déployés.

Dans le même temps, la sécurité du firmware d'un dispositif embarqué moyenne est empiriquement démontré que souvent faible [115, 198]. Cette observation a été souvent faite par des évaluations indépendantes [58, 129, 132, 162]. Ces évaluations confirment souvent que la sécurité de nombreux dispositifs embarqués et leur firmware est très faible. Cela prouve encore une fois que de nombreux vendeurs sont généralement plus intéressés dans la version la plus rapide et le moins cher de nouveaux produits et fonctionnalités pour augmenter leur part de marché. Cette pratique est généralement opposée à la construction de produits sécurisés, où des tests précis est effectué contre les menaces de sécurité actuelles et futures. Ces faits sont d'autant plus préoccupante que les failles de sécurité dans les dispositifs embarqués et leur firmware sont souvent trouvés par les

praticiens de la sécurité en utilisant des approches qui ne sont ni systématiques ni automatisé [36, 121].

Même plus, les vulnérabilités dans le firmware constituent un point d'entrée facile pour les logiciels malveillants et font les dispositifs embarqués sujettes à des attaques simples mais dévastatrices. En fait, depuis 2009, plusieurs botnets (réseaux d'ordinateurs zombies) ont été découverts qui exploitait diverses vulnérabilités de firmware. Ces botnets ont compromis milliers, sinon des millions, de dispositifs embarqués en ligne [49, 65–67, 99, 174, 195, 196]. Pire encore, les dispositifs embarqués affecté sont difficiles à diagnostiquer et à récupérer (par exemple, manque de solutions anti-virus pour les systèmes embarqués, aucune entrée/sortie conventionnelle). Par conséquent, ils restent souvent exploitées pendant de longues périodes de temps. Par exemple, le botnet `Carna` [65] qui a été utilisé pour produire le fameux "Internet Census 2012" ("Recensement Internet 2012") était opérationnel depuis plus d'un an. En plus de cela, le taux de dispositifs embarqués prévu pour connecter à l'Internet est exponentielle et la vitesse des attaques propage à travers les systèmes et réseaux est inimaginable. Par exemple, le virus `Slammer` a infecté plus de 90% des machines vulnérables dans les 10 minutes [158]. En conséquence, l'intervention manuelle ou une analyse est difficile, voire impossible. Cela confirme la nécessité pour détecter les vulnérabilités de firmware avant qu'elles ne soient exploitées par des attaquants. Une analyse manuel de firmware peut trouver ces problèmes [121], mais il peut être beaucoup plus efficace pour automatiser le processus. Dans ce contexte, il est souhaitable que l'analyse de la sécurité du firmware être automatisée et rapide, et être effectuée en continu et à une grande échelle.

La situation devrait devenir encore plus préoccupante pour de multiples raisons, qui peuvent être expliquées à l'aide d'une analyse récente de l'évolution attendue de l'IoT [86]. D'abord, en 2017, le nombre d'appareils connectés IoT devrait dépasser le nombre de PC, les tablettes et les téléphones combinés, et le nombre global de dispositifs IoT installés étant environ 7,5 milliards d'appareils. Deuxièmement, les dispositifs IoT sont prévus pour être à environs également répartie (par exemple, par le comte de l'appareil) entre les secteurs des entreprises, des gouvernements (par exemple, les infrastructures critiques) et les maisons [86]. Cela signifie que tous les grands secteurs sont censés être exposés à des menaces de sécurité provenant de dispositifs embarqués vulnérables. Il est intéressant de noter que le scénario prévu est quelque peu semblable à la hausse dans les attaques mobiles et les logiciels malveillants dans la fin de 2011 [154]. Ce qui est arrivé à peu près quand le nombre d'appareils mobiles en usage (par exemple, smartphones, tablettes) a dépassé le nombre de PC [86]. Il est très probable qu'en 2017 les dispositifs embarqués et IoT devront faire face à des attaques similaires et un examen de la sécurité des technologies mobiles de façon rencontrés en 2011. Toutefois, cela peut probablement se produire à une échelle beaucoup plus grande et ayant un impact élevé.

On sait que les techniques actuelles ne sont pas tout à fait adéquat pour dé-

couvrir efficacement vulnérabilités de firmware de manière évolutive. Certaines techniques, telles que `Avatar` [203], nécessitent souvent l'accès physique aux dispositifs embarqués et la configuration manuelle laborieuse pour chaque appareil. D'autres techniques, telles que `Firmalice` [184], exiger une politique de sécurité doit être fourni pour chaque appareil. Par conséquent, en utilisant les méthodes actuelles, le travail manuel est presque toujours nécessaire et les études à grande échelle sont irréalisables.

En outre, afin de parvenir à une automatisation complète et une amélioration continue du processus d'analyse, deux autres étapes doivent de préférence être automatisées. Premier, que plusieurs fichiers de firmware sont développés et donc recueilli, la possibilité de classifier précisément eux ou dire firmware de non-firmware devient part importante. Par exemple, cela peut être utile de regrouper automatiquement les fichiers du firmware pour le même appareil ou du même fournisseur, puis de les analyser en utilisant des méthodes spécifiques à l'appareil ou le fournisseur. Malheureusement, les efforts actuels pour détecter et classifier les fichiers (par exemple, les logiciels malveillants) [43, 145, 180], ou d'utiliser Machine Learning (ML) [182, 191], sont limitées à des domaines spécifiques et ne couvrent pas spécificités de dispositifs embarqués et leur firmware. Deuxièmement, comme de plus en plus dispositifs embarqués deviennent activé pour le Web et relié à Internet, la capacité d'empreintes digitales et de les identifier avec une grande précision devient important. Par exemple, cela peut être utile pour identifier rapidement et isoler des populations de dispositifs embarqués touchés par une vulnérabilité particulière. Cependant, les techniques actuelles ne peuvent pas être facilement appliqués [60, 90, 113] aux dispositifs embarqués qui sont activé pour le Web ou connectés à l'Internet. Par conséquent, même si l'automatisation de ces deux étapes ne sont pas explicitement liés à l'analyse de la sécurité et de la découverte des vulnérabilités, elles sont souhaitables dans une installation automatisée à grande échelle.

Toutes les considérations ci-dessus sont la base de la nécessité croissante de techniques automatisées à grande échelle pour atteindre deux objectifs principaux. Premier, il consiste à effectuer une analyse de la sécurité effective du firmware. Deuxièmement, il est à classifier avec précision les dispositifs embarqués en ligne et les fichiers du firmware recueillies.

A.1.1 Contributions de la Thèse

Cette thèse décrit des techniques évolutives pour découvrir des vulnérabilités dans le firmware embarqué et de classifier les fichiers de firmware et dispositifs embarqués en ligne. La méthode que nous proposons repose sur des mécanismes automatisés et flexibles. La première étape est de recueillir goulûment et en permanence un grand nombre de fichiers du firmware hétérogènes. Ensuite, nous avons développé des techniques pour décompresser efficacement le firmware et

d'analyser statiquement les fichiers décompressés. Nous essayons aussi d'émuler chaque firmware et ses services (y compris les interfaces Web intégrées) dans une "meilleur effort" et de manière générique. Par la suite, nous avons appliqué l'analyse dynamique (par exemple, les outils de sécurité des applications web) sur chaque instance émulé du firmware. Une étape supplémentaire est la corrélation des vulnérabilités courantes parmi la population des fichiers du firmware. Enfin, nous avons appliqué Machine Learning (ML) pour classifier les fichiers du firmware, et nous avons utilisé les empreintes digitales d'application Web pour classifier les dispositifs embarqués en ligne. La méthodologie de bout en bout est représenté dans la Figure 6.6.

Comme d'autres méthodologies de découvrir et de classifier les vulnérabilités, notre technique ne garantit pas à fournir une couverture complète de vulnérabilités. Pas plus qu'il ne garantit pas une classification tout à fait exact des fichiers du firmware et dispositifs embarqués. Néanmoins, il est la première démonstration de la faisabilité de la découverte de la vulnérabilité dans les fichiers du firmware à grande échelle. Par conséquent, nous prétendons qu'il est un moyen efficace pour aider à accroître la sécurité des dispositifs embarqués, et donc de l'IoT. Même si nous limitons notre travail principalement à la découverte de vulnérabilités dans les images du firmware basé sur Linux (pour les architectures ARM, MIPS et MIPSEL), les mêmes techniques peuvent être facilement étendues à d'autres architectures CPU (par exemple, PowerPC), systèmes d'exploitation (par exemple, VxWorks) et distributions de logiciels. En outre, d'autres méthodes ou des outils d'analyse pourrait être facilement intégré à notre cadre. Par exemple, des outils pour l'exécution symbolique ou fuzzing-outils pourraient être utilisés pour effectuer une analyse dynamique avancé.

Nous avons élaboré un cadre entièrement automatisé, et nous l'avons utilisé pour tester la découverte de vulnérabilités à grande échelle. Notre système a été en mesure de trouver statiquement 38 nouvelles vulnérabilités dans 693 fichiers du firmware. En plus de cela, notre système a été en mesure de découvrir dynamiquement 225 vulnérabilités à impact élevé (par exemple, injection de commandes, XSS) dans au moins 20% des interfaces Web intégrées émulés (45 fichiers de firmware). Nous avons également utilisé notre système pour classifier les fichiers du firmware et les dispositifs en ligne. Notre système automatisé était capable de classifier correctement les fichiers de firmware et d'identifier dispositifs embarqués en ligne avec une précision de 90% ou plus.

Les contributions de cette thèse peuvent être résumées comme suit:

- Nous sommes les premiers à proposer et effectuer la collecte et l'analyse de la sécurité du firmware des dispositifs embarqués à grande échelle.
- Nous avons formulé les cinq premiers défis fondamentaux associés à ce type de recherche, à savoir: Construire un dataset représentatif du firm-

ware; Identification de la firmware; Déballage du firmware et formats personnalisés; Limites d'évolutivité et de calcul; La confirmation des résultats.

- Nous sommes également le premier à démontrer la faisabilité d'automatiser entièrement l'analyse dynamique du firmware hétérogènes à grande échelle. Nous démontrons cela avec une analyse dynamique à grande échelle des interfaces Web intégrées.
- En outre, nous sommes les premiers à proposer une classification précise des dispositifs de firmware et embarqués en ligne, réalisée à grande échelle. Pour cela, nous appliquons Machine Learning (ML), les empreintes digitales de l'interface web et la fusion multi-métrique de scores.
- Nous mettons en œuvre les méthodes proposées dans un cadre entièrement automatisé qui nous a permis de trouver rapidement un grand nombre de nouvelles vulnérabilités dans de nombreux firmware pour une variété de classes de périphériques et les vendeurs.
- Enfin, nous proposons la collecte de firmware, le déballage et l'analyse comme un service en ligne (`http://firmware.re`).

A.1.2 Organisation de la Thèse

Le reste de la thèse est organisé comme décrit ci-dessous:

- Dans le Chapitre 2 nous examinons l'état de l'art appropriés à cette thèse. Nous fournissons une description de la publications, des outils et des expériences existantes présentées par le milieu universitaire et l'industrie.
- Dans le Chapitre 3 nous présentons une étude de cas de l'analyse de bout en bout de la (in) sécurité des systèmes pyrotechniques sans fil. Nous utilisons cette étude de cas comme un exemple de motivation pour notre travail principal dans les chapitres suivants.
- Dans le Chapitre 4 nous présentons notre méthodologie et nous décrivons le système d'analyse à grande échelle que nous avons implémenté. Egalement, nous fournissons les premières indications sur les résultats et les défis d'une telle recherche.
- Dans le Chapitre 5, basé sur le système que nous avons mis en place et nous avons présenté, nous avons mis l'accent sur l'analyse des interfaces web de dispositifs embarqués utilisant l'émulation du firmware combinée à des techniques d'analyse statique et dynamique. Nous montrons comment notre système peut être utilisé dans la pratique pour trouver rapidement de nouvelles vulnérabilités dans les interfaces web de dispositifs embarqués.

- Dans le Chapitre 6 nous nous concentrons sur la classification automatisée et précise des fichiers du firmware et les dispositifs embarqués en ligne. Nous présentons notre expérience avec l'exploration des ensembles possibles de caractéristiques et de mesures d'empreintes digitales. Nous discutons également les applicataions de Machine Learning (ML) et des techniques de fusion de scores multi-métriques pour la classification automatisé et précis à grande échelle.
- Enfin, le Chapitre 7 conclut la dissertation et présente d'autres améliorations éventuelles pour cette thèse.

A.2 Exemple Motivant – Insécurité des Systèmes Pyrotechniques Sans Fil

A.2.1 Introduction

Feux d'artifice et spectacles pyrotechniques sont essentiellement explosifs utilisés principalement pour le divertissement. Un *événement de feux d'artifice*, également appelé un *spectacle pyrotechnique*, est un affichage des effets produits par les appareils du feux d'artifice. Les appareils du feux d'artifice sont conçus pour produire des effets tels que le bruit, la lumière, la fumée, les matériaux flottante (par exemple, des confettis). Les spectacles pyrotechniques et les appareils du feux d'artifice sont contrôlées par *des systèmes de mise à feu*. Les systèmes de mise à feu, en plus des feux d'artifice, sont souvent utilisés à d'autres fins, telles que la démolition des bâtiments, des effets spéciaux, la formation militaire et la simulation militaire.

Malgré le fait que des feux d'artifice sont prévus pour les fêtes, leur utilisation est souvent associée à des risques élevés de destruction, des blessures et même la mort. Beaucoup de nouvelles récente et des études de recherche montrent les dangers de feux d'artifice [17, 172]. Parfois même des feux d'artifice sont utilisés comme de vraies armes dans les combats de rue [33]. Accidents de feux d'artifice sont souvent causés par une mauvaise manipulation du matériel, ne pas suivre les règles de sécurité ou la faible qualité des artifices. Un autre facteur aggravant est que les feux d'artifice sont généralement destinés à être affiché dans les espaces publics qui sont surpeuplés. Tous ces accidents se produisent malgré le contrôle strict de la distribution de feux d'artifice et la nécessité pour une licence professionnelle pour gérer de tels dispositifs.

Classiquement *les systèmes de mise à feu pour les feux d'artifice* composent de commutateurs mécaniques ou électriques et le câblage électrique (souvent appelé "shooting wire"). Ce type de configuration est simple, efficace et relativement sûr [24]. Toutefois, il limite considérablement l'effet, la complexité et les capacités des systèmes de feux d'artifice et des événements pyrotechniques.

Les progrès de logiciels / firmware, dispositifs embarqués et des technologies sans fil permet aux systèmes de feux d'artifice de profiter pleinement de ces technologies. Un système (sans fil) de feux d'artifice moderne peut être considéré comme un bon exemple d'un *système cyber-physique embarqué (Embedded Cyber-Physical System (ECPS))* ou *réseau sans fil des capteurs/actionneurs (Wireless Sensor/Actuator Network (WSAN))*. Les systèmes de mise à feu des feux d'artifice compter de plus en plus sur les technologies sans fil, embarqués et de logiciels/firmware. Par conséquent, ils sont exposés aux mêmes risques que tout autre ECPS, WSAN ou systèmes informatiques.

Basé sur des recherches récentes, systèmes d'infrastructures critiques et systèmes embarqués de tous types acquis une mauvaise réputation en matière de sécurité. Par exemple, les avions peuvent être falsifiés sur les nouveaux systèmes de surveillance radar [76], le contrôle de la voiture peut être détourné [69, 148] ou peut être exploitée à l'échec [126], une pompe à insuline implantée peut être complètement attaqué [173] ou un réseau de contrôleurs logiques programmables (Programmable Logic Controllers (PLC)) dans une installation nucléaire peut être rendu non fonctionnel [109, 150].

Dans ce chapitre, nous étudions les risques des systèmes de mise à feu du point de vue de la sécurité des systèmes embarqués, des réseaux sans fil et sécurité informatique. Nous détaillons notre expérience de découvrir et exploiter un système de mise à feu sans fil dans un court laps de temps sans aucune connaissance préalable de ces systèmes. En bref, nous démontrons notre méthodologie à partir de l'analyse du firmware à la découverte de vulnérabilités. Notre analyse statique a aidé notre décision d'acquérir un tel système qui nous avons analysé plus en détail. Cela nous a permis de confirmer la présence de vulnérabilités exploitables sur le périphérique réel. Enfin, nous insistons sur la nécessité de la sécurité de l'appareil et le logiciel/firmware, et sur l'application de la conformité de la sécurité pour les systèmes de mise à feu pyrotechniques.

A.2.2 Sommaire

Nous avons présenté la découverte de la vulnérabilité et l'exploitation des systèmes de mise à feu sans fil dans un court laps de temps sans connaissance préalable de ces systèmes. Nous avons commencé avec un système automatisé à grande échelle pour la récupération et l'analyse du firmware (détaillé dans le Chapitre 4). Dans cette expérience, nous avons utilisé des heuristiques simples (par exemple, mot-clé correspondant) et l'analyse statique très simple. Cela nous a permis d'isoler rapidement et automatiquement les fichiers de firmware des systèmes de mise à feu d'importance fondamentale. Nous avons également été en mesure d'identifier plusieurs vulnérabilités potentielles en utilisant l'analyse statique automatique et manuel. Ces vulnérabilités comprennent la mise à jour de firmware non authentifié, les communications sans fil non authentifiés,

l'espionnage et les communications sans fil de type spoofing, injection de code arbitraire, le déni de service temporaire. Nous avons implémenté et testé avec succès une attaque simpliste avec des conséquences potentiellement dévastatrices.

Nous concluons que, compte tenu du risque présenté par leur utilisation, la sécurité des systèmes de mise à feu sans fil doit être prise très au sérieux. Nous concluons également que de tels systèmes doivent être plus rigoureusement réglementés et certifiés. Nous insistons sur la nécessité et l'urgence d'introduire vérification de la conformité des logiciels et du matériel similaire à DO-178B et DO-254, respectivement. Nous croyons fermement ces petites étapes d'amélioration, ainsi que des solutions à la Section 3.3.5, peut certainement contribuer à accroître la sécurité et la sûreté de ces systèmes embarqués sans fil.

Dernier point, mais non des moindres, nous avons discuté des problèmes avec le vendeur. Une mise à jour du firmware, qui est maintenant déployé, corrige la plupart des les problèmes de sécurité. Malheureusement, il ya plus de 20 fournisseurs de systèmes pyrotechniques sans fil qui peuvent rester vulnérables à des attaques similaires, en particulier certains d'entre eux ne disposent d'un mécanisme de mise à jour du firmware.

Dans ce chapitre, nous avons démontré comment l'analyse de la sécurité peut être effectuée sur un seul appareil en utilisant principalement une analyse manuelle. Bien que cette analyse est très utile pour découvrir les problèmes de sécurité graves dans les systèmes embarqués, cette approche est pas extensible. Dans le reste de cette thèse, nous allons présenter les techniques pour automatiser certaines des étapes du processus d'analyse de la sécurité pour dispositifs embarqués.

A.3 Analyse à Grande Échelle de la Sécurité des Firmwares pour Dispositifs Embarqués

A.3.1 Introduction

Les systèmes embarqués sont omniprésents dans notre vie quotidienne et sont de plus en de plus présents dans nombreux environnements informatiques en réseau. Par exemple, ils sont à la base de divers dispositifs Common-Off-The-Shelf (COTS) tels que les imprimantes, les routeurs, composants et périphériques pour ordinateurs. Ils sont également présents dans de nombreux dispositifs qui sont moins axées sur les consommateurs tels que les systèmes de surveillance vidéo, les implants médicaux, des éléments de voiture, systèmes industriels SCADA et PLC, et pratiquement tout ce que nous appelons communément les appareils *électroniques*. Le phénomène émergent de l'Internet des objets (Internet-of-Things (IoT)) va rendre ces systèmes encore plus communs et interconnectés.

Tous ces systèmes exécutent un logiciel spécial, souvent appelé *firmware*, qui est généralement distribué par les fournisseurs comme *des fichiers de firmware* ou *des mises à jour firmware*. Plusieurs définitions de *firmware* existent dans la literature. Le terme a été introduit à l'origine pour décrire le microcode du processeur (CPU) qui existait "quelque part entre" les couches de matériel et le logiciel. Cependant, le mot a rapidement acquis un sens plus large. La norme IEEE 610.12-1990 [35] étendu la définition de couvrir le *"combinaison d'un dispositif de matériel informatique et instructions ou des données qui résident sur le périphérique matériel comme logiciel en lecture seule"*.

À l'heure actuelle, le terme *firmware* est plus généralement utilisé pour décrire le logiciel qui est incorporé dans un dispositif matériel. Comme les logiciels traditionnels, firmware pour dispositifs embarqués peut avoir des bugs ou des erreurs de configuration qui peuvent entraîner des vulnérabilités pour les dispositifs qui exécutent ce code particulier. Basé sur des preuves anecdotiques, les systèmes embarqués ont acquis une mauvaise réputation en matière de sécurité, généralement basé sur cas par cas des expériences d'échecs. Par exemple, la commande de l'accélérateur d'une voiture échoue [126] ou peuvent être malicieusement détourné [69, 148]; un routeur domestique sans fil se trouve à avoir une backdoor [36, 121, 132], pour ne citer que quelques exemples récents. D'une part, en dehors de quelques projets qui ciblé des dispositifs ou des versions de logiciels spécifiques [82, 116, 181], à ce jour il n'y a toujours pas d'analyse de sécurité à grande échelle de fichiers du firmware. D'autre part, l'analyse manuel de la sécurité des fichiers du firmware rendements des résultats très précis, mais il est extrêmement lent et n'est pas évolutive pour un ensemble de données vaste et hétérogène des fichiers du firmware. Aussi utile que ces rapports individuels sont pour un dispositif ou la version de firmware particulier, ces rapports seuls ne permettent pas d'établir un jugement général sur l'état général de la sécurité du firmware. Pire encore, la même vulnérabilité peut être présent dans des dispositifs différents, qui sont laissées vulnérables jusqu'à ce que ces défauts sont re-découvert indépendamment par d'autres chercheurs [132]. Ceci est souvent le cas lorsque plusieurs *fournisseurs d'intégration* compter sur les mêmes sous-traitants, des outils, des kits de développement logiciel ou fournies par *fournisseurs de développement*. De nombreux appareils peuvent aussi être marqués sous des noms différents, mais peuvent effectivement exécuter la même ou similaire firmware. De tels dispositifs sont souvent affectés par exactement les mêmes vulnérabilités, cependant, sans une connaissance approfondie des relations internes entre les fournisseurs, il est souvent impossible d'identifier ces similitudes. En conséquence, certains appareils sont souvent laissés affectés par les vulnérabilités connues, même si une mise à jour firmware est disponible.

Contributions

En bref, ce chapitre apporte les contributions suivantes:

- Nous démontrons les avantages d'effectuer une analyse à grande échelle de firmware fichiers et nous décrivons les principaux défis liés à cette activité.
- Nous proposons un système pour effectuer des prélèvements de firmware, le filtrage, le déballage et l'analyse à grande échelle.
- Nous avons mis plusieurs techniques statiques efficaces que nous avons appliquées sur $32,356$ fichiers firmware.
- Nous présentons une technique de corrélation qui permet de propager les informations de vulnérabilité aux fichiers du firmware similaires.
- Nous avons découvert 693 fichiers de firmware touchés par au moins une vulnérabilité et nous avons signalé 39 nouveaux vulnérabilité (Common Vulnerabilities and Exposures (CVE)).

A.3.2 Sommaire

Dans ce chapitre, nous avons présenté une analyse statique à grande échelle de fichiers de firmware pour dispositifs embarqués. Nous avons montré que une vue plus large sur le firmware non seulement est bénéfique mais également est effectivement nécessaire pour la découverte et l'analyse des vulnérabilités des dispositifs embarqués. Notre étude aide les chercheurs et les analystes de sécurité de mettre la sécurité des dispositifs particuliers dans leur contexte, et leur permet de voir comment les vulnérabilités connues qui se produisent dans un firmware réapparaissent dans le firmware des autres fabricants. Les ensembles de données résumées sont disponibles à `http://firmware.re/usenixsec14`.

Dans les deux prochains chapitres suivants, nous décrivons plusieurs améliorations à notre système. Dans le Chapitre 5, nous essayons d'émuler le fichiers du firmware en exécutant le firmware désarchivée l'intérieur de l'émulateur QEMU. Nous faisons cela pour permettre une analyse statique et dynamique évolutive. Nous montrons l'efficacité de l'amélioration en effectuant une analyse évolutive des interfaces web au sein de plusieurs centaines de fichiers du firmware. Ensuite, dans le Chapitre 6, nous appliquons Machine Learning (ML) pour classifier et étiqueter les fichiers du firmware inconnus. Nous utilisons également des multi-score de fusion pour la classification des empreintes digitales au niveau du HTTP de dispositifs embarqués en ligne. Grâce à ces améliorations, nous adressons partiellement les défis de "Construire un ensemble de données représentant", "Identification du firmware", "Limites d'évolutivité et de calcul" et "Confirmation des résultats" tel que présenté dans la Section 4.2.

A.4 Analyse Dynamique de Firmware à Grande Échelle: Une Étude de Cas sur les Interfaces Web de Dispositifs Embarqués

A.4.1 Introduction

Au cours des quelques dernières années, les appareils embarqués sont devenus plus connecté formant ce qu'on appelle l'Internet des objets (IdO, IoT). Ces dispositifs sont souvent mis en ligne par la composition; la fixation d'une interface de communication à un dispositif (non sécurisé) existant. La plupart de ces dispositifs manquent de l'interface utilisateur des ordinateurs de bureau (par exemple, clavier, vidéo, souris), mais doivent néanmoins être administré. Quoique certains dispositifs reposent sur des protocoles personnalisés tels que clients "thick" ou même les interfaces existantes (à savoir, telnet), le web est rapidement devenu "de facto" le interface universelle d'administration. Par conséquent, le firmware de ces appareils a souvent intégré un serveur web qui exécute des applications web simples à assez complexes. Pour le reste de ce chapitre, nous allons nous référer à ces comme *interfaces web de dispositifs embarqués*.

Il est bien connu que la sécurisation des applications web est une tâche difficile. En particulier, les chercheurs ont montré que plus de 70% des vulnérabilités sont hébergés dans la couche des applications (web) [171]. Les attaquants, qui sont familiers avec cette réalité, ils utilisent une variété de techniques pour exploiter des applications web. Vulnérabilités bien connues, telles que SQLinjection [62] ou Cross Site Scripting (XSS) [199], sont encore fréquemment exploités et constituent une partie importante des vulnérabilités découvert chaque année [71]. En outre, les vulnérabilités telles que Cross Site Request Forgery (CSRF) [45], command injection [188], et HTTP response splitting [142] sont également très souvent présent dans les applications web.

Compte tenu d'un tel palmarès des problèmes de sécurité dans les systèmes embarqués et les applications web, il est naturel de s'attendre à la pire des interfaces web de dispositifs embarqués. Cependant, comme nous le verrons, ces vulnérabilités ne sont pas faciles à découvrir, analyser et confirmer.

Vue d'Ensemble de Notre Approche

Afin d'effectuer des tests scalable de la sécurité des interfaces web de dispositifs embarqués, nous avons développé un système d'analyse automatisé (Figure 5.1). Nous avons commencé notre analyse par un ensemble de données de 1925 fichiers de firmware qui ont été précédemment décompressé [1] qui contiennent des interfaces web. Ensuite, pour chaque firmware déballé nous identifions des structures

[1] Nous nous sommes concentrés principalement sur les fichiers du firmware basé sur Linux. Fichiers du firmware basé sur Linux sont en général bien structuré et documenté,

de documents web potentiels présents à l'intérieur du firmware. À ce stade, nous faisons une passe avec des outils d'analyse statique sur les structures de documents web. Ensuite, nous effectuons l'émulation des fichiers du firmware. Quand (et si) le serveur web est en fonctionnement, la phase d'analyse dynamique est effectué. Enfin, nous analysons les résultats et effectuer une analyse manuelle chaque fois que nécessaire.

Contributions

En bref, ce chapitre fait les contributions suivantes:

- Nous effectuons la première étude complète de sécurité sur les interfaces web de dispositifs embarqués, à grande échelle. Nous faisons cela en misant sur plusieurs techniques et outils de état de l'art.
- Nous détaillons une partie de firmware précédemment non étudiée, et nous découvrons de sérieuses vulnérabilités dans un large spectre de dispositifs embarqués.
- Nous proposons une méthodologie efficace et nous développons un système scalable pour aborder la détection de vulnérabilités dans les interfaces web de dispositifs embarqués.
- Nous permettons un banc d'essai pour de nouvelles recherches de sécurité avancée sur le firmware des systèmes embarqués.

A.4.2 Sommaire

Dans ce chapitre, nous avons présenté une nouvelle méthode pour effectuer une analyse à grande échelle de la sécurité des interfaces web au sein de dispositifs embarqués. À cette fin, nous avons conçu un système qui utilise des logiciels de série pour l'analyse statique et dynamique. En raison des limitations dans les outils d'analyse statique, nous avons créé un mécanisme pour l'émulation automatique des fichiers de firmware. Alors que l'émulation du matériel parfaitement inconnue restera probablement une question ouverte, nous étions en mesure d'émuler les systèmes assez bien pour tester les interfaces web de 246 fichiers de firmware. Notre système a trouvé de sérieuses vulnérabilités dans au moins 24% des interfaces web que nous étions en mesure d'émuler. En incluant la phase d'analyse statique, 9290 problèmes ont été trouvées sur un total de 185 fichiers de firmware. Ceci comprend 225 vulnérabilités à *impact élevé* que nous avons pu vérifier.

Enfin, nos expériences et les résultats confirment que la sécurité d'un grand nombre de ces dispositifs est sérieusement défaut. Nous visons donc à l'exécution

par conséquent, ils sont plus faciles à déballer, analyser et emuler. Cependant, notre approche peut être facilement étendu à l'avenir à d'autres types de firmware, y compris les firmwares monolithiques.

de ce système comme un processus continu. Cela peut aider à améliorer la qualité et de continuer à trouver des vulnérabilités dans ces dispositifs embarqués, en espérant qu'ils seront corrigés par les vendeurs. Un tel service pourrait également être utile aux fournisseurs de dispositifs embarqués qui peuvent bénéficier de tests de sécurité automatique avant d'expédier leurs produits. Nous espérons que notre système peut aider à rendre l'Internet et l'IoT plus sûr et sécurisé.

A.5 Classification des Firmware et l'Identification des Appareils Embarqués Dans une Manière Scalable

A.5.1 Introduction

Un fichier du firmware est, en général, fait sur mesure pour un dispositif embarquée spécifique, et un modèle d'appareil contient et exécute un fichier de firmware particulier. Cela est relativement facile pour une personne à suivre lors de l'analyse manuelle (par exemple, le processus de mise à jour du firmware). Cependant, parce que les appareils embarqués sont si diverses, il est pas trivial pour les ordinateurs et les systèmes automatisés pour relier un modèle d'appareil embarqué et un fichier du firmware.

Par exemple, lorsque téléchargement manuellement un fichier de firmware à partir d'un site du fournisseur, il est souvent relativement facile pour une personne de connaître le vendeur et le périphérique pour lequel le firmware est destiné. Cependant, pour un système automatisé qui télécharge des milliers de fichiers du firmware à partir de sites non structurées, il est pas une tâche triviale pour classifier les fichiers du firmware par classe de périphérique ou même par le vendeur. Nous avons identifié et décrit ce problème comme le défi de la "Identification du Firmware" dans le Chapitre 4.

Problèmes Ouverts

Dans ce contexte, nous avons identifié et nous avons formulé deux problèmes comme suit: Tout d'abord, comment étiqueter *automatiquement* et *avec précision* la marque et le modèle de l'appareil pour lequel le firmware est destiné. Deuxièmement, comment identifier *automatiquement* et *avec précision* le vendeur, le modèle et la version du firmware d'un dispositif embarqué arbitraire en ligne.

Ces mesures doivent être effectuées d'une manière fiable, indépendant du dispositif, le vendeur, ou des protocoles personnalisés qui exécutent sur l'appareil.

Vue d'Ensemble de Notre Approche

Dans notre méthode, nous appliquons Machine Learning (ML) pour classifier les fichiers du firmware selon les fournisseurs des dispositifs embarqués et des types de tels dispositifs. Nous utilisons deux algorithmes largement adoptés, Random Forests (RF) et Decision Trees (DT), sur la base de leur mise en œuvre dans le logiciel `scikit-learn` [168]. Nous explorons plusieurs ensembles de caractéristiques dérivées des caractéristiques des fichiers du firmware, tels que la taille du fichier, l'entropie du fichier et les chaînes de caractères communes. Ensuite, nous recommandons les caractéristiques optimales définies pour ce type de problèmes de classification et montrent que notre approche permet d'obtenir une grande précision de la classification. En outre, en utilisant des méthodes statistiques solides tels que les intervalles de confiance, nous estimons la performance de nos classificateurs pour des données réelles à grande échelle.

Nous recueillons alors empreintes digitales des interfaces web sur les périphériques réels et sur les périphériques émulés basés sur les fichiers du firmware précédemment classifiés. Nous construisons une base de données d'empreintes digitales sur la base de ces périphériques émulés et réels. Ensuite, nous pouvons correspondre interfaces web d'un appareil embarqué inconnue à la liste des empreintes digitales web connus dans notre base de données en utilisant plusieurs métriques correspondants, tels que le plan du site (sitemap) ou le Finite-State Machine (FSM) du protocole HTTP. Enfin, nous utilisons plusieurs systèmes de notation pour classifier les correspondances d'empreintes digitales.

Contributions

En bref, ce chapitre fait les contributions suivantes:

- Nous appliquons Machine Learning (ML) dans le contexte de la classification des fichiers du firmware, et nous proposons et étudions les caractéristiques du firmware qui rend cela possible.
- Nous montrons que l'utilisation de Machine Learning (ML) permet de classifier automatiquement des ensembles de fichiers du firmware avec une grande précision.
- Nous étudions la prise d'empreintes digitales et d'identification des dispositifs embarqués et leur version du firmware en utilisant des empreintes digitales multi-métrique des interfaces web de dispositifs embarqués (physiques et émulé).

A.5.2 Sommaire

Dans ce chapitre, nous avons présenté deux techniques complémentaires, à savoir *classification supervisée des fichiers du firmware* et *identification par em-*

preintes digitales des interfaces web de dispositifs embarqués. Nous avons proposé Machine Learning (ML) pour le défi de la classification des firmwares et nous avons exploré la fusion multi-métrique de scores pour le problème de l'identification des interfaces web de dispositifs embarqués. Avec une grande confiance pour les données réel et à grande échelle, nos tests démontrent que les classificateurs et caractéristiques nous proposons peut atteindre la précision de 93.5% pour la classification du firmware et la précision de 89.4% pour l'identification des dispositifs embarqués.

A.6 Conclusions

Cette thèse a présenté de nouvelles techniques pour automatiser l'analyse à grande échelle de la sécurité du logiciel dans les systèmes embarqués et de leur firmware. Nous avons mis ces techniques dans un système complète et validée son efficacité avec des données réelles (i.e., les fichiers de firmware et les dispositifs embarqués en ligne).

Nous avons commencé le voyage de cette thèse avec un système simple mais efficace. Les modules de téléchargement de notre système collectées efficacement les fichiers du firmware pour des fins d'analyse ultérieure. Nous avons utilisé les robots simples basées sur une combinaison de "site scrapers", les requêtes de recherche personnalisé et téléchargeurs de fichiers à partir des pages de support technique. Nous avons pu recueillir 172k fichiers de firmware potentiels. Nos estimations ultérieures ont montré que, avec un confiance de 95% il devrait y avoir au moins $34\% \pm 8\%$ de véritables fichiers de firmware dans notre ensemble de données. Cette base de données alors nécessaire déballage et analyse de la sécurité. Les modules de déballage de notre système sont basées sur une extension personnalisée de Binary Analysis Toolkit (`BAT`) , mais il peut aussi être facilement étendu à l'avenir avec d'autres systèmes de déballage, par exemple, `binwalk`. Même si les modules déballage ne peuvent pas encore garantir déballage complet de tout firmware, ils déballés environ 75% des fichiers du firmware traitées. En fin de compte, les modules d'analyse statique simples nous ont permis de trouver 38 nouvelles vulnérabilités dans 693 fichiers du firmware. Par exemple, ces modules font analyse de mots de passe faibles en `/etc/passwd`, ils recueillent et ils suivent les clés privées en ligne (SSL, SSH), et ils vérifient les erreurs de configuration de sécurité simples et les backdoors évidentes. Ces expériences nous ont permis d'identifier cinq défis importants associés à l'analyse à grande échelle de la sécurité du firmware des dispositifs embarqués. Par ailleurs, certains défis ont été fortement liée à des zones insuffisamment étudiés que nous avons exploré avec succès par la suite.

Nous avons ensuite amélioré notre système par l'introduction de l'analyse statique et dynamique automatisé et à grande échelle des interfaces web de dispositifs embarqués. L'approche est basée sur l'émulation des systèmes de fichiers

de firmwares (extrait lors du déballage des fichiers du firmware), d'une manière distribuée et indépendant de l'architecture. Nous avons aussi développé et utilisé un ensemble d'heuristiques automatisés pour augmenter la réussite tant de l'émulation du firmware et du lancement de l'interface web de dispositifs embarqués. Dès que les interfaces web de dispositifs embarqués sont lancés, notre système applique des outils d'analyse statique et dynamique sur eux. Notre système entièrement automatisé découvert en seulement quelques heures vulnérabilités à impact élevé (par exemple, injection de commandes, XSS) dans au moins 20% des interfaces web émulés de dispositifs embarqués. Dans le même temps, il pourrait automatiquement émuler les interfaces web de dispositifs embarqués dans les 15% des fichiers du firmware contenant des interfaces web. L'augmentation de ces taux de réussite de manière automatisée et intelligente est un défi que nous voulons aborder dans nos travaux futurs. Notre approche est flexible ce qui signifie que de nouvelles techniques d'émulation et d'analyse peuvent être facilement ajoutés dans le futur. En outre, si l'on sait maintenant que de nombreux dispositifs embarqués ne sont pas sécurisés, notre système est vraiment la première démonstration de la possibilité de pleinement et faisable automatiser l'analyse dynamique du firmware hétérogène de dispositifs embarqués à grande échelle.

Enfin, nous avons amélioré notre système avec l'intelligence supplémentaire en employant Machine Learning (ML) et des techniques de classification. Pour classifier les fichiers du firmware, nous avons exploré les algorithmes de Random Forests (RF) et Decision Trees (DT) en combinaison avec plusieurs ensembles de caractéristiques. Sur notre ensemble de données du firmware, nous avons montré que l'algorithme de RF avec les caractéristiques `[size, entropy, entropy extended, category strings, category unique strings]` est le meilleur choix parmi les quatre principaux ensembles de fonctionnalités que nous explorées. Par exemple, notre système atteint plus de 90% de précision de la classification lorsque les ensembles de formation ont été basées sur au moins 40% de chaque catégorie de firmware connu. Pour classifier les dispositifs embarqués en ligne, nous avons exploré les empreintes digitales des interfaces web en utilisant des techniques de fusion multi-métrique de scores. Notre système repose sur les empreintes digitales des interfaces web de dispositifs embarqués, calculé sur six métriques. Ensuite, il classe les métriques d'empreintes digitales à l'aide de trois systèmes d'évaluation, et utilise la technique de la fusion des scores dans l'évaluation finale de la meilleure correspondance des empreintes digitales. Nous avons aussi raisonnablement motivés nos choix pour les métriques et les systèmes d'évaluation dans le cadre d'interfaces web de dispositifs embarqués. Par exemple, en moyenne, notre système atteint 89.4% de précision dans l'identification de l'appareil en utilisant une base de données de 31 empreintes digitales des interfaces web de dispositifs embarqués. Enfin, nous avons démontré qu'il est possible de classifier les fichiers de firmware et d'identifier dispositifs embarqués en ligne avec une grande précision.

A.6.1 Les Travaux Ultérieurs

Les travaux ultérieurs sera axée sur la construction d'ensembles de données propres, annotées et représentatives de fichiers du firmware et d'émulations de dispositifs. Par exemple, ces ensembles de données pourraient être utilisés par les praticiens comme données "vérité-terrain" ("ground truth") pour les expériences ultérieures. Cela permettrait d'évaluer l'efficacité et l'efficience de nouvelles techniques visant à découvrir des vulnérabilités dans les systèmes embarqués et de leur firmware. En outre, cela permettrait une comparaison juste et raisonnable entre les différentes techniques et approches.

Une autre amélioration prévue pour les travaux futurs consiste à exploiter un mécanisme de CAPTCHA pour construire de manière incrémentale un ensemble bien marqué de fichiers du firmware pour former les classificateurs de Machine Learning (ML). Par exemple, les fichiers de firmware non classés pourraient être présentées aléatoirement pour la classification à plusieurs utilisateurs de notre service en ligne http://firmware.re. Pour la validation croisée de la réponse des utilisateurs, un fichier du firmware avec un bon label connu est également présentée comme un défi aux utilisateurs ainsi que ceux non classés. Un défi dans ce processus pourrait être la conception d'une représentation visuelle convaincante des fichiers du firmware présentés aux utilisateurs. Finalement, fois que un fichier de firmware non classés atteint un seuil de classification dans une catégorie particulière, il est ajouté à l'ensemble de données d'entraînement sous l'étiquette de la catégorie.

Nous prévoyons également de développer des outils et des techniques pour l'analyse statique qui sont nouveaux, avec un accent particulier sur leurs applications à la sécurisation des programmes dans les fichiers du firmware. Par exemple, ceux-ci pourraient être des outils d'analyse statique pour les technologies du web qui ne sont pas bien couverts par l'état de l'art, tels que Lua ou Haserl. Dans d'autres cas, ceux-ci pourraient être des méthodes d'analyse statique binaires pour la myriade d'architectures CPU moins communs trouvés dans les dispositifs embarqué et dans les dispositifs de l'IoT.

Enfin, nous prévoyons de développer, déployer et surveiller "honeypots" robustes et réalistes au maximum émulant dispositifs embarqué hétérogènes. Cela permettrait de capturer et d'analyser à des stades précoces de nouvelles menaces, les exploits et les logiciels malveillants qui ciblent un large éventail de dispositifs embarqués et de dispositifs pour l'IoT.

Ethical Aspects

Large-scale scans testing 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.

In our study we were particularly careful to work within legal and ethical boundaries. Below we summarize the steps we took to ensure that we do not inadvertently cross those boundaries.

- We obtained 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.
- We strictly followed the *responsible disclosure* policy. In this regard, we tried our best to notify vendors, CERTs and Vulnerability Contribution Programs (VCP) for vulnerabilities we discovered during our experiments. We also tried to assist vendors in reproducing these issues.
- We did not involve any device in our main methodology. This avoids both accessing devices we did not own and breaching terms of use. Also, there was no risk to interfere unintentionally with devices which were not under our control or to "brick" an actual device. In limited cases when confirmation of an issue required a physical device, we performed such validations on devices under our control and in an isolated test environment.
- 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.

List of Figures

List of Tables

Bibliography

[1] https://github.com/travisgoodspeed/goodfet. 28

[2] http://w3techs.com/technologies/overview/programming_language/all. 61

[3] http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html. 61

[4] http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList. 61

[5] http://rips-scanner.sourceforge.net. 61

[6] http://code.google.com/p/rough-auditing-tool-for-security. 61

[7] http://www.scovetta.com/yasca.html. 61

[8] http://www.arachni-scanner.com/. 62

[9] https://code.google.com/p/zaproxy/. 62

[10] http://w3af.org/. 62

[11] http://owasp.org/index.php/Top_10_2013-A1-Injection. 62

[12] http://www.darrinhodges.com/chroot-voodoo/. 66

[13] http://justniffer.sourceforge.net/. 71

[14] http://nmap.org. 75

[15] Atmel AppNote AVR411: Secure Rolling Code Algorithm for Wireless Link. 22

[16] Audit PHP Configuration Security Toolkit. 55

[17] California Fireworks Display Goes Horribly Wrong: Dozens injured by catastrophic misfire during Simi Valley Fourth of July. ABCNews, 5th July 2013. 19, 110

[18] CVE-2007-1435, CVE-2011-4821. 75

[19] CVE-2010-2965, CVE-2014-0659. 75

[20] CVE-2014-4880, CVE-2013-1606. 75

[21] CVE-2014-9222. 75

[22] Define of backdoor string in DLink DI-524 UP GPL source code. https://gist.github.com/ccpz/6960941. 52

[23] Econotag. http://redwire.myshopify.com/. 29

[24] Fireworks Electric (Wired) Firing Systems. http://www.skylighter.com/fireworks/how-to/setup-electric-firing-systems.asp. 19, 22, 110

[25] Google Custom Search Engine API. 42

[26] Internet Census 2012 – Port scanning /0 using insecure embedded devices. http://internetcensus2012.bitbucket.org. 7, 8

[27] KillerBee; Framework and tools for exploiting ZigBee and IEEE 802.15.4 networks. http://code.google.com/p/killerbee/. 29

[28] Microchip SST25VF032B Flash Chip Datasheet. 25

[29] Motorola ColdFire MCF52254 Processor Datasheet. 25

[30] NFPA 79: Electrical Standard for Industrial Machinery. http://www.nfpa.org/79. 22

[31] Synapse Module Comparison Chart. http://content.solarbotics.com/products/documentation/synapse_comparison_table.pdf. 24, 26

[32] Synapse SNAP Network Operating System – Reference Manual, v2.4, 2012. 26

[33] Ukraine protests: Kiev fireworks 'rain on police'. http://www.bbc.com/news/world-europe-25820899. 19, 110

[34] USB Snap Stick SS200. https://www.synapse-wireless.com/snap-components/usb-mesh-snap-stick. 25, 28

[35] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. 33, 113

[36] Slashdot: Backdoor found in TP-Link routers, March 2013. 1, 34, 106, 113

[37] Download statistics for the wemo android application, February 2014. http://xyo.net/android-app/wemo-JJUZgf8/. 52

[38] Download statistics for the wemo iOS application, February 2014. http://xyo.net/iphone-app/wemo-J1QNimE/. 52

[39] P. Alvarez. Using Extended File Information (EXIF) File Headers in Digital Evidence Analysis. *International Journal of Digital Evidence*, 2(3):1–5, 2004. 15

[40] K. Andersson and P. Szewczyk. Insecurity by obscurity continues: are adsl router manuals putting end-users at risk. 2011. 11

[41] ANONYMIZED. ANONYMIZED (under submission). 98

[42] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2011. 71

[43] 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 Recent Advances in Intrusion Detection (RAID)*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag. 2, 17, 107

[44] 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 *IEEE Symposium on Security and Privacy*, 2008. 14, 55, 58

[45] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *ACM Conference on Computer and Communications Security (CCS)*, 2008. 57, 115

[46] 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. 10

[47] 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. 16, 47

[48] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy*, 2010. 14, 58, 62

[49] T. Baume. Netcomm nb5 botnet–psyb0t 2.5 l. Technical report, Technical report, January 2009. http://www. adam. com. au/bogaurd/PSYB0T. pdf, 2009. 1, 11, 106

[50] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *ISOC Network and Distributed System Security Symposium (NDSS)*, NDSS '09. The Internet Society, 2009. 17

[51] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A View on Current Malware Behaviors. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, LEET'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association. 38

[52] B. Bencsath, L. Buttyán, and T. Paulik. XCS Based Hidden Firmware Modification on Embedded Dievices. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2011. 14, 74

[53] C. M. Bishop et al. *Pattern recognition and machine learning*, volume 4. springer New York, 2006. 87

[54] A. Blanco and M. Eissler. One firmware to monitor'em all. *Ekoparty*, 2012. 10

[55] BlindElephant. Web Application Fingerprinter. `http://blindelephant.sourceforge.net`, Jun 2015. 15

[56] A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial intelligence*, 97(1):245–271, 1997. 87

[57] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross Channel Scripting and Its Impact on Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2009. 14, 74

[58] H. Bojinov, E. Bursztein, E. Lovett, and D. Boneh. Embedded management interfaces: Emerging massive insecurity. *BlackHat USA*, 2009. 1, 8, 14, 58, 105

[59] D. Bongard. Fingerprinting Web Application Platforms by Variations in PNG Implementations. *Blackhat*, 2014. 15

[60] K. Bonne Rasmussen and S. Capkun. Implications of Radio Fingerprinting on the Security of Sensor Networks. In *International Conference on Security and Privacy in Communications Networks (SecureComm)*, 2007. 3, 16, 89, 107

[61] J.-Y. L. Boudec. *Performance Evaluation of Computer and Communication Systems*. EFPL Press, 2011. 49, 87

[62] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*, 2004. 57, 115

[63] M. Brocker and S. Checkoway. iSeeYou: Disabling the MacBook webcam indicator LED. In *USENIX Security Symposium*, 2013. 10

[64] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. volume 8, pages 209–224, 2008. 12

[65] Carna-Botnet. Internet census 2012: Port scanning/0 using insecure embedded devices, 2013. 1, 2, 7, 11, 106

[66] P. Čeleda, R. Krejčí, and V. Krmíček. Flow-based security issue detection in building automation and control networks. In *Information and Communication Technologies*, pages 64–75. Springer, 2012. 1, 11, 106

[67] P. Čeleda, R. Krejčí, J. Vykopal, and M. Drašar. Embedded malware-an analysis of the chuck norris botnet. In *Computer Network Defense (EC2ND), 2010 European Conference on*, pages 3–10. IEEE, 2010. 1, 11, 106

[68] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM. 47

[69] 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 *USENIX Security Symposium*, SEC'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association. 20, 33, 111, 113

[70] K. Chen. Reversing and exploiting an Apple firmware update. *BlackHat USA*, 2009. 10

[71] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. *Mitre Report*, 2007. 57, 115

[72] Cisco. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. Apr 2011. 1, 105

[73] A. Costin. Hacking Printers for Fun and Profit. 10, 37

[74] A. Costin. PostScript(um): You've Been Hacked. 10, 37

[75] A. Costin. All your cluster-grids are belong to us: Monitoring the (in)security of infrastructure monitoring systems. In *1st IEEE Workshop On Security and Privacy in the Cloud*. IEEE, 2015. xx

[76] A. Costin and A. Francillon. Ghost in the Air (Traffic): On insecurity of ADS-B protocol and practical attacks on ADS-B devices. *Black Hat USA*, July 2012. xx, 20, 23, 111

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

[78] A. Costin, J. Isacenkova, M. Balduzzi, A. Francillon, and D. Balzarotti. The role of phone numbers in understanding cyber-crime schemes. In *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*, pages 213–220. IEEE Computer Society Press, 2013. xx

[79] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A large scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*. USENIX, 2014. xx

[80] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. *arXiv*, (arXiv:1511.03609), 2015. xix

[81] A. Cui. Embedded Device Firmware Vulnerability Hunting with FRAK. *DefCon 20*, 2012. 9, 43, 44, 127

[82] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013. 10, 34, 37, 43, 113

[83] A. Cui and S. J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-area Scan. In *Annual Computer Security Applications Conference (ACSAC)*, 2010. 7, 8, 16, 34

[84] Z. Cutlip. Emulating and Debugging Workspace. `http://shadow-file.blogspot.fr/2013/12/emulating-and-debugging-workspace.html`. 77

[85] J. Dahse and T. Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014. 14, 55, 58, 61, 73, 74, 127

[86] T. Danova. The internet of everything. *Business Insider, Feb*, 28, 2014. 2, 106

[87] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security Symposium*, SEC'13, pages 463–478, Berkeley, CA, USA, 2013. USENIX Association. 12, 13

[88] 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. 47

[89] G. Delugré. Closer to metal: reverse-engineering the Broadcom NetExtreme's firmware. *Hack.lu*, 2010. 10

[90] L. C. C. Desmond, C. C. Yuan, T. C. Pheng, and R. S. Lee. Identifying Unique Devices Through Wireless Fingerprinting. In *ACM conference on Wireless network security*, 2008. 3, 16, 89, 107

[91] B. F. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable Reverse Engineering for the Greater Good with PANDA. 2014. 13

[92] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012. 87

[93] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)*, 2011. 58

[94] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *USENIX Security Symposium*, 2012. 14

[95] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2010. 14, 68

[96] L. Duflot, Y.-A. Perez, and B. Morin. Netcraft. PHP Usage Stats. `http://www.php.net/usage.php`, June 2007. 73

[97] L. Duflot, Y.-A. Perez, and B. Morin. What If You Can't Trust Your Network Card? In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011. 10

[98] K. Dunham. A fuzzy future in malware research. *The ISSA Journal*, 11(8):17–18, 2013. 38

[99] L. Durfina, J. Kroustek, and P. Zemek. Psybot malware: A step-by-step decompilation case study. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 449–456. IEEE, 2013. 1, 11, 106

[100] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, et al. The matter of Heartbleed. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 475–488. ACM, 2014. 9

[101] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman. Analysis of the HTTPS Certificate Ecosystem. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*, IMC '13, pages 291–304, New York, NY, USA, 2013. ACM. 8, 51, 53

[102] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX Security Symposium*, 2013. 8, 47, 53, 74

[103] P. Eckersley. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies Symposium (PETS)*, 2010. 16

[104] K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Proceedings of the Network & Distributed System Security Symposium, San Diego, CA*, 2012. 30

[105] 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. 55

[106] F. B. et al. QEMU – Quick EMUlator. http://www.qemu.org. 59

[107] eurialo. Lightidra IRC Router Scanner – Lightaidra is an IRC commanded tool that allows for scanning and exploiting routers. https://packetstormsecurity.com/files/109244, 2012. 11

[108] D. Ewing. Synapse's snap network operating system. http://www.synapse-wireless.com/upl/downloads/industry-solutions/reference/white-paper-synapse-snap-network-operating-system-96f6130b.pdf. 24

[109] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. *White paper, Symantec Corp., Security Response*, 2011. 20, 111

[110] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, 2010. 58

[111] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1, Jun 1999. 91

[112] E. Fong and V. Okun. Web Application Scanners: Definitions and Functions. In *Annual Hawaii International Conference on System Sciences (HICSS)*, 2007. 14, 58

[113] J. Franklin, D. McCoy, P. Tabriz, V. Neagoe, J. V. Randwyk, and D. Sicker. Passive Data Link Layer 802.11 Wireless Device Driver Fingerprinting. In *USENIX Security Symposium*, 2006. 3, 16, 89, 107

[114] G. "Fyodor" Lyon. NMAP (Network Mapper) – a free and open source utility for network discovery and security auditing. http://nmap.org/. 7, 96

[115] D. Geer. Cybersecurity as Realpolitik. *BlackHat*, 2014. 1, 105

[116] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward Secure Embedded Web Interfaces. In *USENIX Security Symposium*, 2011. 14, 34, 113

[117] C. Heffner. binwalk – firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images. 9, 43

[118] C. Heffner. Emulating NVRAM in Qemu. http://www.devttys0.com/2012/03/emulating-nvram-in-qemu/. 77

[119] C. Heffner. littleblackbox – Database of private SSL/SSH keys for embedded devices. 50

[120] C. Heffner. Breaking SSL on Embedded Devices, December 2010. 50

[121] C. Heffner. Reverse Engineering a D-Link Backdoor, October 2013. 1, 2, 34, 35, 52, 106, 113

[122] D. Hely, F. Bancel, M.-L. Flottes, and B. Rouzeyre. Secure scan techniques: a comparison. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, pages 6–pp. IEEE, 2006. 30

[123] 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. 10, 43

[124] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, 2012. 8

[125] V. Hilderman and T. Baghi. *Avionics certification: a complete guide to DO-178 (software), DO-254 (hardware)*. 2007. 23

[126] J. Hirsch and K. Bensinger. Toyota settles acceleration lawsuit after $3-million verdict. Los Angeles Times, October 25, 2013. 20, 33, 111, 113

[127] H. Holm, T. Sommestad, J. Almroth, and M. Persson. A quantitative evaluation of vulnerability scanning. *Information Management & Computer Security*, 19(4):231–247, 2011. 14

[128] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and Precise Sanitizer Analysis with BEK. In *USENIX Security Symposium*, 2011. 58

[129] HP-Fortify-ShadowLabs. Report: Internet of Things Research Study. http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA5-4759ENW, 2014. 1, 9, 105

[130] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *International Conference on World Wide Web (WWW)*, 2003. 58

[131] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on World Wide Web (WWW)*, 2004. 14

[132] Independent Security Evaluators. SOHO Network Equipment (Technical Report), 2013. 1, 8, 34, 35, 105, 113

[133] Intel. Rise of the Embedded Internet. 2009. 1, 105

[134] IOActive. Critical DASDEC Digital Alert Systems (DAS) Vulnerabilities, June 2013. 50

[135] IOActive. stringfighter – Identify Backdoors in Firmware By Using Automatic String Analysis, May 2013. 52

[136] IOActive. Critical Belkin WeMo Home Automation Vulnerabilities, February 2014. 50

[137] J. Isacenkova, O. Thonnard, A. Costin, D. Balzarotti, and A. Francillon. Inside the scam jungle: A closer look at 419 scam email operations. In *Security and Privacy Workshops (SPW), 2013 IEEE*, pages 143–150. IEEE, 2013. xx

[138] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM. 47

[139] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006. 14, 55, 61, 73

[140] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010. 58

[141] M. Kammerstetter, C. Platzer, and W. Kastner. PROSPECT – Peripheral Proxying Supported Embedded Code Testing. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014. 13, 58

[142] A. Klein. Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks and Related Topics. *Sanctum whitepaper*, 2004. 57, 115

[143] L. A. Klein. *Sensor and data fusion: a tool for information assessment and decision making*, volume 324. Spie Press Bellingham, 2004. 94

[144] T. Kohno, A. Broido, and K. C. Claffy. Remote Physical Device Fingerprinting. *Dependable and Secure Computing, IEEE Transactions on*, 2(2):93–108, 2005. 16, 89, 96

[145] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 7:2721–2744, 2006. 2, 107

[146] J. Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digit. Investig.*, 3:91–97, 2006. 16, 38

[147] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006. 93

[148] 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 *IEEE Symposium on Security and Privacy*, SP '10, pages 447–462, Washington, DC, USA, 2010. IEEE Computer Society. 20, 33, 111, 113

[149] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems. In *USENIX Workshop on Offensive Technologies (WOOT)*, Washington, D.C., Aug. 2015. USENIX Association. 13

[150] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011. 20, 111

[151] H. Li, D. Tong, K. Huang, and X. Cheng. FEMU: A Firmware-Based Emulation Framework for SoC Verification. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2010. 13

[152] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008. 45

[153] B. Livshits and S. Chong. Towards Fully Automatic Placement of Security Sanitizers and Declassifiers. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2013. 58

[154] D. Maslennikov. Mobile Malware Evolution, Part 5. 2012 [cited 2012 26 March]. 2, 106

[155] J. Matherly. SHODAN – Computer Search Engine. http://www.shodan.io. 8, 47, 74

[156] P. C. Messina, R. D. Williams, and G. C. Fox. *Parallel computing works !* Parallel processing scientific computing. Morgan Kaufmann, San Francisco, CA, 1994. 39

[157] C. Miller. Battery firmware hacking. *BlackHat USA*, 2011. 10

[158] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security & Privacy*, (4):33–39, 2003. 2, 106

[159] H. Moore. Security Flaws in Universal Plug and Play: Unplug, Don't Play. *Rapid7, Ltd.*, 2013. 8

[160] Morning Star Security. WhatWeb. http://www.morningstarsecurity.com/research/whatweb, Jun 2015. 15

[161] K. Nandakumar, Y. Chen, S. C. Dass, and A. K. Jain. Likelihood ratio-based biometric score fusion. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2):342–347, 2008. 94

[162] M. Niemietz and J. Schwenk. Owning Your Home Network: Router Security Revisited. In *Web 2.0 Security and Privacy (W2SP) Workshop*, 2015. 1, 9, 16, 90, 105

[163] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *IEEE Symposium on Security and Privacy*, 2013. 16, 89

[164] Nvidia. CUDA – Compute Unified Device Architecture Programming Guide. 2007. 45

[165] OpenwallProject. John the Ripper password cracker. http://www.openwall.com/john/. 45

[166] OWASP. Top 10 Vulnerabilities, 2013. 51

[167] 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)*, Washington, D.C., Aug. 2015. USENIX Association. 12

[168] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011. 80, 118

[169] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-Architecture Bug Search in Binary Executables. In *IEEE Symposium on Security and Privacy*, San Jose, CA, May 2015. 12

[170] Postscapes. Internet of Things Market Forecast. 2014. 1, 105

[171] J. Prescatore. Gartner, quoted in ComputerWorld, 2005. 57, 115

[172] V. Puri, S. Mahendru, R. Rana, and M. Deshpande. Firework injuries: a ten-year study. *Journal of Plastic, Reconstructive & Aesthetic Surgery*, 62(9), 2009. 19, 110

[173] J. Radcliffe. Hacking medical devices for fun and insulin: Breaking the human scada system, August 2011. http://cs.uno.edu/~dbilar/BH-US-2011/materials/Radcliffe/BH_US_11_Radcliffe_Hacking_Medical_Devices_WP.pdf. 20, 111

[174] B. Rodrigues. Analyzing Malware for Embedded Devices: TheMoon Worm. http://w00tsec.blogspot.fr/2014/02/analyzing-malware-for-embedded-devices.html, 2014. 1, 11, 106

[175] V. Roussev. Data Fingerprinting with Similarity Digests. In *IFIP Int. Conf. Digital Forensics*, pages 207–226, 2010. 16, 38, 39

[176] W. Salusky and M. E. Thomas. Patent US8244799 – Client Application Fingerprinting Based on Analysis of Client Requests, Aug 2012. 15, 89

[177] W. Salusky and M. E. Thomas. Patent US8694608 – Client Application Fingerprinting Based on Analysis of Client Requests, Apr 2014. 15

[178] M. Samuel, P. Saxena, and D. Song. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*, 2011. 58

[179] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2011. 58

[180] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001. 2, 107

[181] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *ACM Conference on Computer and Communications Security (CCS)*, 2013. 12, 34, 61, 113

[182] A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE, 2010. 2, 107

[183] S. Shah. HTTP Fingerprinting and Advanced Assessment Techniques. *Blackhat*, 2003. 15, 90

[184] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice: Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015. 2, 12, 61, 107

[185] 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. 52

[186] S. Stamm, Z. Ramzan, and M. Jakobsson. Drive-by pharming. In *9th International Conference on Information and Computer Security (ICICS)*. Springer Berlin Heidelberg, Aug. 2007. 10

[187] J. V. Stough. distributed-python-for-scripting – DistributedPython for Easy Parallel Scripting. 45

[188] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006. 57, 115

[189] Symantec. IoT Worm Used to Mine Cryptocurrency. http://www.symantec.com/connect/blogs/iot-worm-used-mine-cryptocurrency, 2014. 12

[190] G. Taleck. Ambiguity Resolution via Passive OS Fingerprinting. In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003. 96

[191] R. Tian, L. Batten, R. Islam, and S. Versteeg. An automated classification system based on the strings of trojan and virus families. In *Malicious and*

Unwanted Software (MALWARE), 2009 4th International Conference on, pages 23–30. IEEE, 2009. 2, 107

[192] Tjaldur Software Governance Solutions. Binary Analysis Tool (BAT). 10, 43

[193] A. Tridgell. rsync – utility that provides fast incremental file transfer. 45

[194] J. B. Ullrich. cmd.so Synology Scanner Also Found on Routers. https://isc.sans.edu/diary/cmd.so+Synology+Scanner+Also+Found+on+Routers/17883, 2014. 12

[195] J. B. Ullrich. Linksys Worm ("TheMoon") Captured. https://isc.sans.edu/forums/diary/Linksys+Worm+TheMoon+Captured/17630, 2014. 1, 11, 106

[196] J. B. Ullrich. Linksys Worm ("TheMoon") Captured. https://isc.sans.edu/diary/Linksys+Worm+%22TheMoon%22+Summary%3A+What+we+know+so+far/17633, 2014. 1, 11, 106

[197] J. B. Ullrich. More Device Malware: This is why your DVR attacked my Synology Disk Station (and now with Bitcoin Miner!). https://isc.sans.edu/diary/More+Device+Malware%3A+This+is+why+your+DVR+attacked+my+Synology+Disk+Station+%28and+now+with+Bitcoin+Miner!%29/17879, 2014. 12

[198] J. Viega and H. Thompson. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security & Privacy*, 10(5):68–70, 2012. 1, 105

[199] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2007. 57, 115

[200] Wappalyzer. Identify Software on the Websites You Visit. https://wappalyzer.com, Jun 2015. 15

[201] D. A. Wheeler. SLOCCount – a set of tools for counting physical Source Lines of Code (SLOC). http://www.dwheeler.com/sloccount/. 48

[202] xobs and bunnie. The Exploration and Exploitation of an SD Memory Card. *CCC – 30C3*, 2013. 53

[203] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014. 2, 12, 30, 58, 77, 107

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

[205] 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 *Annual Computer Security Applications Conference (ACSAC)*, ACSAC '13, pages 279–288, New York, NY, USA, 2013. ACM. 10, 37

[206] 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. 15

[207] M. Zheng, M. Sun, and J. Lui. Droidray: a security evaluation system for customized android firmwares. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 471–482. ACM, 2014. 8