UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Analysing the IOT threat landscape using consumer firmware-based, high fidelity honeypots

Mihàly Csonka

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

2021
2022

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Analysing the IOT threat landscape using consumer firmware-based, high fidelity honeypots

Mihàly Csonka

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

Universiteit Hasselt

A thesis submitted in partial fulfilment of the requirements for the degree Master of Science in Computer Science

Analysing the IoT threat landscape using consumer firmware-based, high fidelity honeypots

Author:

Mr. Mihály Csonka

Advisor:

Prof. Peter Quax

Co-advisor:

Prof. Wim Lamotte

Supervisors:

Mr. Isaac Meers

Academic year 2021–2022

Acknowledgments

First of all, I would like to thank my counsellor Mr. Isaac Meers and co-advisor Prof. Wim Lamotte for their extensive feedback and assistance. Also, I thank my advisor Prof. Peter Quax for the possibility of writing my thesis under his department. Without their guidance I would not have been able to complete this project.

Mr. Niels Van Belle's technical assistance was of great help in the later stages of this thesis. Furthermore, the department of Computer Sciences sponsored the hosting of the honeypots. I wish to thank them for their support.

I am also grateful to all other professors and (assistant) teachers whom I met during my whole educational career. Thank you for sharing your knowledge, encouraging me to keep asking questions, and keep studying.

An additional thank you goes out to my great friends for putting up with my daily stupidity and obsession with my hobbies.

And last but not least, my family at home. Thank you for giving me so many opportunities in life and supporting me along the whole way. I am really lucky and grateful to have you.

— Mihály

Abstract

The total amount of IoT devices has long since outgrown the human population. Yet, just as any other technology, hackers abuse it for their personal benefit. Cyber attacks involving IoT devices are notorious for their devastating effects. Understanding the underlying cause and working of these attacks is paramount in order to defend ourselves against similar abuse in the future. As it turns out, hardware capabilities, human knowledge, and financial incentives negatively affect the development process of IoT devices. This results in security issues that were considered solved 20 years ago. However, not only the devices, but several popular IoT protocols also prove to be inherently vulnerable.

In order to study the techniques and malware binaries currently being used by hackers, data needed to be collected on them. However, we could not risk the compromise of production network infrastructure. As such, a network filled with various fake IoT devices was built with the intent of tricking hackers into accessing it. We argue that no one would be fooled by the network unless it contained some consumer devices. Thus, a comparison of several techniques to host consumer devices was made. Unfortunately, as we found out, none of the techniques are perfect for our goal. Half of them are unusable to emulate IoT devices realistically, while the other half have their own distinct advantages and disadvantages. Nevertheless, to achieve the goal of building said believable, yet fake, IoT network, an informed decision was made to choose one of the latter techniques.

After having collected data for a few months, we came to several conclusions. The first of which is that there is not much diversity in the world of IoT malware. Most attacks, and malware binaries, follow the same patterns. The reason being that they build upon the code of older malware that was made public. Furthermore, IoT attacks are highly automated and generic. This allows them to spread as fast, and as wide, as they do. Default credentials and not performing updates prove to be the two major malpractices that allow hackers to gain access to a device. Then, the malware's focus lies on accessing internet-facing devices, rather than spreading through private networks that they gained access to via infected devices. And lastly, while they are certainly functional, IoT device emulation techniques require more research and development in order to become viable in this area of security research.

Contents

Chapter 1

Introduction

Internet of Things (IoT) is a rapidly growing paradigm connecting everyday objects to the Internet. Be it cameras, thermostats, routers, or fridges, IoT devices have been appearing everywhere from homes to schools, hospitals, and businesses in the recent years. Despite growth being impacted by a worldwide silicon shortage and the COVID-19 pandemic, the amount of connected devices in 2021 grew by a billion compared to the year before. The total amount is predicted to keep growing significantly over the next decade [Sin21]. Enhancing one's life with IoT has become the future's way of living. The devices assist us humans in our daily chores and as such attempt to improve our quality of life. Yet, the paradigm is as notorious as it is popular. Every single device becomes a possible victim of abuse by being connected to the Internet. Combine this with the huge attack surface that is the result of both malpractices and hardware constraints and it is quickly clear why some people despise the thought of using any such device. The integration with everyday objects makes IoT devices ever so prevalent in our lives. This results in not only vast amounts of data, but also highly sensitive data being gathered by the devices' various sensors. Such information can include maps of various rooms in one's house, daily living patterns, video/audio recordings, movie interests, etc. Possible misuse, by hackers and corporations alike, means a privacy nightmare is waiting to happen [Gie19; Ren+19].

Abuse of IoT devices could physically inconvenience users, such as in 2017 when a hacker caused printers exposed to the Internet to print messages warning their owners of the exposure [Cim17]. A step further would be harming both device and user. For example, excessive power consumption by malware can result in the device's battery being depleted and the device thus being unable to function properly. This could quickly go south if the device is being relied on for safety purposes, such as a fire alarm or a lock. Lastly, individual IoT devices do not provide much in terms of computing power. Yet, the sheer amount of devices allows for the building of a network with unparalleled amounts of resources. The Mirai botnet was one of the first of its size using IoT devices. It gathered attention by taking down services such as cybersecurity blogs [Kre16a], hosting providers [OVH16], and DNS service providers [Int16] with Distributed Denial-of-Service (DDoS) attacks. Despite said gathered attention, IoT security has not notably improved since Mirai in 2016 as numerous alternative botnets with even higher throughput have since wrecked havoc [GY21].

Security research on the paradigm has been growing along with both users' and adversaries' interest in it. Among others, this includes the study of communication protocols [Kay+20; Bou+20], libraries [San+21], and hardware [Shu+19] specific to IoT devices. Vulnerability research by trusted sources is valuable in that it allows vendors to produce appropriate patches. Unfortunately, unless these patches are applied, devices will stay vulnerable. An alternative path of research is thus gathering data on attacks seen "in the wild". This is in order to understand the actual day to day threats to which devices are exposed. The information can then be used by both vendors and network administrators to improve defensive technologies such as Intrusion Detection Systems (IDS) and firewalls. One way of obtaining this information is by analysing the artifacts left on production resources after being exposed to an attack. Aside from being clearly undesirable, it is also a non-scientific method as logging mechanisms might not cover the attack in enough detail. The adversary might even tamper with artifacts in order to leave no trails behind. An alternative, controlled way of capturing adversaries' behaviour is by using honeypots. These are systems designed to be attacked while looking like production resources. This makes them a complementary technique to the traditional defensive applications such as IDSs and firewalls [Fan+18a; Mai+11].

From a survey on IoT honeypots performed by Franco et al., it can be concluded that numerous honeypots with focus on imitating IoT services have been created in the past decade [Fra+21]. Nevertheless, the majority of these projects either only offer a single service endpoint, or provide adversaries with a minimal, imperfect shell environment. Although a handful of higher fidelity projects exist, these are not fully reproducible due to lack of source code or ethical considerations, or do not use consumer firmware. Additionally, most research projects focus on single devices, despite interconnectedness being one of the strengths of IoT. Covering the above shortcomings is the contributions made by this thesis. In other words, we create a consumer firmware-based, high fidelity honeypot that exposes several IoT-related services and is placed in a network along with other IoT honeypots.

Building a high fidelity honeypot that provides adversaries with a real consumer device's environment is complex. Virtualising parts of an IoT device comes with complications that are the result of the characteristics that make IoT devices exactly so diverse and popular. That is, being designed for a specific task and interacting with the physical world. Approaches that nevertheless do attempt this are called firmware re-hosting techniques. No prior work exists that compares these techniques in the context of building a honeypot. As such, as an additional contribution, we make this comparison.

The goal of this thesis is the analysis of malicious interactions with IoT devices. A high fidelity IoT honeypot is built and connected to the Internet in order to gather sufficiently verbose data for this analysis. Unique credentials, along with assisting information, is shared online in order to lure adversaries. The honeypot simulates embedded devices in a home environment, with network depth. Meaning, an adversary is given the opportunity to move through the network. The goal of the honeypot is to be scalable, i.e. requiring no hardware and being able to be deployed on any host, and to host consumer firmware, all the while having the standard characteristics of a honeypot, e.g being believable. Virtualising consumer firmware requires firmware re-hosting. However, this technology is still young and unrefined, and has only been used once in literature to build a honeypot [VC19]. As such, this thesis presents a case study. The intention of the thesis, discussed above, can be defined with following research questions:

1. What techniques and/or malware do adversaries employ during attacks involving IoT devices?
2. Given that IoT devices are made to solve specific problems and thus differ wildly, to what extent do adversaries adapt to their targets. In other words, are the employed techniques and/or malware targeted?
3. Do adversaries attempt lateral movement through an IoT network, and how effective are they at it?
4. Is state of the art firmware re-hosting usable to build a believable, high fidelity IoT honeypot?

We now describe the structure of the rest of this thesis. Chapter 2 is a literature study of IoT. This includes an exploration of what IoT entails, why IoT devices are more vulnerable than traditional devices, and a structural overview of IoT attacks. The workings and vulnerabilities of several application layer communication protocols, which are popular in the IoT world, are also studied. Second, a literature study of honeypots is performed in chapter 3. This chapter not only describes the components and properties, but also the requirements of a functional honeypot. To finish the literature study, IoT and honeypots are brought together. In an attempt to find an appropriate technique to host consumer firmware, several approaches are explored and compared by us. Chapter 4 discusses our implementation along with design decisions and encountered complications. Last but not least, the captured data is analysed in chapter 5, and chapter 6 concludes this thesis.

Chapter 2

Internet of Things (IoT)

The universal interest surrounding IoT has resulted in rapid economical growth, which in turn fuels development. Despite this, or maybe in spite of it, definitions are manifold. IoT is also being employed in various fields. This leads to each device being unique in functionality and design. In order to better understand this inherently heterogeneous concept, how IoT fits into the physical world is explored first. Then, different types of systems are discussed based on how their firmware is designed. Tackling the issue of security, we study how the inherent characteristics of the IoT paradigm impact security. Next, IoT malware is dissected based on prior work. Lastly, some application level protocols that are commonly used by IoT devices are discussed. We look at their design, high-level working, and potential security flaws.

2.1 Informed environment or industrial control?

At the most basic level, IoT can be defined as interconnected embedded devices. Embedded devices, in turn, being everyday objects enhanced by embedding special purpose computers that are designed to perform a highly specific task. By embedding them in everyday objects, the computers are brought close to the source of information and target to be affected. In this thesis, the term IoT device is interpreted as such. Yet, as embedded devices and the IoT ecosystem evolved over time, alternate definitions arose. For example, ENISA, The European Union Agency for Network and Information Security, defines IoT as "a cyber-physical ecosystem of interconnected sensors and actuators, which enable intelligent decision making" [ES21]. Not only does this definition not mention Internet-connectedness, as one would expect given the acronym's literal meaning, it also uses the words "cyber-physical [...]system", which make up the acronym CPS.

While the IEEE attempts to discuss and standardise the evolving definitions of connected devices [IEE15], Greer et al. argues that many of the definitions are already the same [Gre+19]. Their interpretation is as follows. IoT and CPS systems, the overarching terms for private and industrial connected devices respectively, connect the digital and physical world. This is achieved with a continuous cycle, shown in fig. 2.1. First, **information** is passed through ICT systems and is processed. This causes **actuators** to influence the physical world. The influence manifests as **energy** flowing through engineered systems, resulting in change in the environment. Lastly, these changes are captured by **sensors**, producing digital information and thus completing the cycle. Humans are tightly coupled with this cycle and can interact with it on numerous levels. For example, they can be part of the engineered system or design the implementation of the cycle.

2.2 Types of systems

Embedded systems are intended to perform specific tasks. Nevertheless, as both requirements and capabilities change over time, so does the design and implementation of systems. Muench et al. categorises devices based on their firmware [Mue+18a]. They argue that vulnerabilities present themselves differently based on how the resources of a system are managed. Wright et al. adopts this categorisation in their analysis of emulation techniques for embedded devices [Wri+21]. Indeed, understanding how the

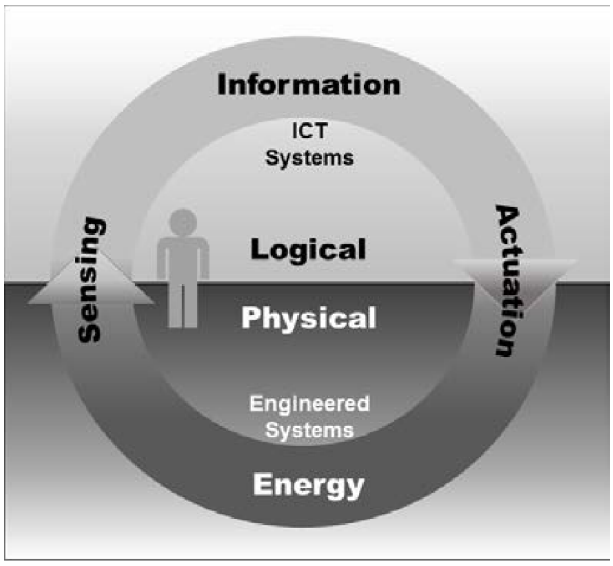


Figure 2.1: The "components model" describes the continuous cycle of interactions in connected device environments. Figure taken from [Gre+19].

firmware works enables the usage of specific techniques and heuristics in the design and implementation of an emulator. Emulation techniques will be discussed in depth in section 3.4. In the rest of this thesis, we will mainly focus on Type 1 devices.

Type 1: general purpose devices employ a generic Operating System (OS). As the name implies, the OS aims to provide a generic environment for software to run in. It manages process, memory, and hardware interactions. The OS is usually a retrofitted version of a pre-existing operating system [Mue+18a; Wri+21]. Changes to e.g. minimise storage usage and to the network stack are made to enable running the OS on low power devices. Development of Type 1 systems, compared to the other types, is becoming more enticing as system complexity and hardware capability increases [Wan17, p.265].

Type 2: special purpose OSes still provide separation between kernel space and user space, similar to a general purpose OS. However, resource managing algorithms have been redesigned or dropped altogether as to optimise for the constraints of embedded devices. Real Time OSes (RTOS) are the most prevalent special purpose OSes. As their name implies, RTOSes are designed to respond to events in (near) real time and process tasks within appropriate time limits. To achieve this, RTOSes minimise interrupt latency, time spent in critical regions by means of optimised memory management, and implement scheduling algorithms that can prioritise tasks based on priority and their supposed deadline [Wan17, p.401]. This is different from general purpose OSes, which simply aim to balance resources fairly over time for all processes.

Type 3: bare metal systems have the highest level of coupling between hardware and software. The device is meant to do a single thing and the software reflects this fact. There is little to no abstraction as there is no OS to manage memory, processes, and hardware interactions, leaving all the above to be handled by custom implementations [Lal13, p.517-519]. Even so, some libraries do exist to handle these and ease development. The firmware is monolithic as it contains both device management and business logic. An example implementation is a simple infinite loop that handles some co-routines sequentially. Bare metal software can be made highly optimised but lacks portability.

2.3 Inherent risk

The concept of IoT combines common technologies that have extensively been scrutinized over the years, and have now seemingly become innocent. Yet, when employed within the context of IoT, they contribute to the notoriety of it. Antonakakis et al. compares the state of IoT security to that of PCs in the 2000s [Ant+17]. The reason being that common best practices are not implemented. This includes protection of binaries by means of e.g. Address Space Layout Randomization (ASLR) and stack canaries, network security such as only exposing required ports, and access control by means of for example the

concept of least privilege and complex passwords. The lack of defensive security measures means that a malware's minimal viable product requires a low amount of effort to create. In other words, the high amount of vulnerable devices and ease of infecting them creates a high return on investment. Supporting this idea are some notable differences between IoT-focused malware and more "traditional" malware, i.e. server and PC focused malware, as will be discussed later in section 2.4.2. This lack of a security conscious design is caused by several inherent characteristics of the paradigm and results in a large attack surface [MS19; Zha+14]. Surveys such as [ES17] and [Nes+19] extensively list the causes of said insecure design in production devices. We group these in an attempt to categorise them. However, note that causes often go hand in hand.

Being embedded in everyday objects means that the used hardware has less **capabilities** than traditional hardware, i.e. computing power, memory, available energy, etc. is limited. These properties influence security in the following ways:

- Physical security: Due to being placed throughout the environment, human supervision is not possible at all times. A physical component accessible by adversaries opens up new attack surfaces. For example, they could cause physical damage, read data such as encryption keys or private user data from a debug port, or even use lasers to inject audio commands into devices with microphones [Sug+20].
- Energy harvesting: Some IoT devices get power from a battery. Excessive usage may empty it fast, which leads to a Denial-of-Service (DoS) attack [Pal20]. This could either be caused by an adversary or by faulty design.
- Encryption: The constrained hardware prevents the implementation of strong encryption algorithms. Such algorithms use a significant amount of resources. Thus, encrypting data with low computing power would, among others, result in non-trivial delays and high energy consumption. While lightweight encryption algorithms are actively being researched, limitations still exist [Sin+17]. Some hardware even lacks an appropriate source of entropy [Men+19]. Encryption is an important security tool that affects, among other things, communication, storage, and authentication. Consequently, weak encryption opens up many avenues to abuse such as impersonation, data injection, user privacy violations, etc.
- Auditing: The creation, storage, and automated processing of events is not evident. Both creation and processing require CPU cycles, which are already limited. Storage space is also limited on constrained devices. Additionally, local storage might not even be an option as some manufacturers employ read-only filesystems [Dan+19; Alr+21]. External storage and processing might be a possibility, but once again requires encryption to ensure confidentiality and integrity of information. A program performing auditing must also be tamper proof to protect its working from being altered by malware.

Developers might not have the required **knowledge** to properly secure the devices. IoT devices combine multiple areas of expertise due to being a bridge between the physical world and the digital world. Despite developers possibly being aware of best practices within one or a few areas, expecting knowledge of all is infeasible. Assigning an expert to each is similarly difficult given the tight vertical coupling, i.e. multi-area expertise, required. Additionally, the integration of different technologies raises additional security challenges. For example, configuring an IoT device via a web interface commonly invokes a system binary using the `system()` function. A developer knowledgeable about web technologies might not have (enough) experience with the Unix shell. Improper sanitisation would result in a command injection vulnerability. The rapid evolution of technology such as cloud integration, lack of generalised device architecture due to e.g. highly specific use cases, and lack of abstraction layers further complicates this. Users might also lack the knowledge to secure their devices as they consider these to be "different" from everyday technological devices. All in all, lack of knowledge manifests in bad programming practices, lacking authentication and access control methods, faulty usage of protocols, infrequent updating, lacking physical security, etc. [ES17]. Indeed, many devices use weak default credentials, hardcoded (`root`) user credentials, and/or do not require users to change the default credentials. Also, devices often lack the proper management of permission. This results in adversaries easily being able to gain access to accounts with high levels of permissions. These simple mistakes were at the core of the Mirai botnet [Ant+17], despite extensive prior research [LXC12; ano12; Oua+17].

Vendors lack the **incentive** to properly secure devices and protocols. Morgner and Benenson argues that consumers care more about accessibility, features, and interoperability than about security [MB18].

This results in a lack of security testing, which in turn amplifies the occurrence of earlier mentioned malpractices. Complex security mechanisms might even completely be ignored to reduce development time. In 2017, the European Commission noted how the proliferation of technology increases the possible impact of attacks on European Union citizens. They mention IoT devices in particular, saying that "cybersecurity is not [...] prioritised in their design" [Eur17]. At the same time, a proposal was submitted that in 2019 resulted in a new regulation [Eur19]. This regulation defines a European cybersecurity certification framework for vendors. Acquiring such certification not only tests and indicates the security level of a product, but requires vendors to give consumers appropriate information including e.g. how to maintain the security of their device. Unfortunately, said framework is voluntary and as such does not contribute enough to raise incentive among vendors.

2.4 Decomposing IoT malware

The ultimate goal of studying attacks is to understand their working. This knowledge can then be used to build defences against future attacks. Indeed, unless a defender knows what to look for, they will not be able to stop an attack. Similarly, automated processes require some indicator to mark an interaction as malicious. In practice, such indicators are either signatures or heuristics. Signatures are highly specific and thus used to identify distinct attacks, or parts thereof. Heuristics, on the other hand, are used to identify more generic patterns in e.g. the behaviour or composition of a binary. This section describes several ways of classifying IoT malware.

2.4.1 File-based vs. fileless malware

Traditionally, malware is thought to be a file that somehow ends up on a system. This file, when executed, will spawn its own process(es) and perform malicious actions such as encrypting the filesystem, add the system to a botnet, or install itself in multiple locations to ensure persistency. File-based malware is popular as all desired functionality can be provided by a single file. However, Mansfield-Devine argues that advances in understanding of file-based malware spurred improvements in detection and prevention techniques [Man17]. For example, warning users when running untrusted software, scanning for unexpected files on the filesystem, or scanning files being sent over the network are effective defensive techniques. Similarly, security researchers are taught to look for unconventional files when performing forensic analysis of an infected system [Kum+20]. In other words, file-based malware has a high chance of being fingerprinted.

In contrast, fileless malware does not require a file being dropped on a victim's filesystem. Instead, adversaries leverage trusted resources such as already installed software, or OS features such as the command-line shell. Malicious code and/or instructions are loaded into memory [Kum+20; Dan+21]. Sometimes fileless malware does use the filesystem, but not in a traditional sense. For example, the Kovter virus adds entries containing shell commands to the Windows registry that are automatically read and executed when certain events occur [San17]. Fileless malware may be slightly harder to create than the alternative, but it has the advantage of a drastically lower fingerprint. Signature-based detection is not as effective for fileless malware. Leveraged trusted resources are by definition not marked suspicious. Additionally, scanning memory is more complicated than scanning the filesystem. Instead, using heuristics to look for certain patterns, e.g. in the behaviour of software by means of process or system log inspection, is more effective [Kum+20].

The effectiveness of file-based malware against traditional computers, e.g. PCs and servers, has decreased over time. Yet, it is still effective against IoT devices. Unsurprisingly, the reason for this is the constrained hardware used by IoT. It complicates running signature-based detection software on the device itself. Network administrators can install e.g. an IDS or firewall with deep packet inspection capability in a business environment. In a home environment, however, usage of such a system is not always feasible. Heuristic-based detection techniques are even more resource intensive than their counterparts. Fileless malware is thus equally hard to combat in constrained devices. Dang et al. found that fileless malware in IoT manifests mainly as only shell commands [Dan+19]. Inspecting log files could thus be a viable defensive technique. Yet, the authors note that in order to prevent file-based malware, some vendors implement read-only filesystems. This hinders auditing. As an alternative, they recommend vendors to remove as many unused executables, i.e. trusted resources, as possible.

2.4.2 The attack lifecycle

Malware attacks are complex, multistage processes. Each malware implementation differs slightly in how the stages are performed. Simpler attacks may even skip some to decrease complexity in exchange for a loss in effectiveness. MITRE has mapped out these stages along with numerous techniques used within each stage. Their framework, called MITRE ATT&CK®, allows one to study adversaries' actions in a structured manner [MIT21]. Although Type 1 devices are Unix-like, the MITRE Linux framework does not apply in full to IoT-focused malware. This is due to how IoT devices differ from more traditional devices, such as PCs and servers. The IoT platform's malpractices and design decisions, discussed in section 2.3, influence the way adversaries approach the attack process. For example, many firmwares do not employ proper access control and instead run everything as the `root` user. The MITRE framework's Privilege Escalation step is thus not applicable to IoT devices.

Palo Alto Networks, Inc. instead identifies eight stages, of which most match or aggregate the stages described by MITRE [Pal20]. Despite not classifying it under the same stages, the data captured with IoT-focused honeypots by the authors of [Pa+15], [Dan+19], and [TOS21] confirm that all but the Lateral Movement stage are common occurrences in practice. Additionally confirming and highlighting the differences with traditional malware, i.e. malware targeting PCs, servers, and mobile devices, Alrawi et al. and Vignau, Khoury and Hallé studied the workings of numerous IoT malware families [Alr+21; VKH19]. They note the usage of a subset of techniques used in traditional malware. Although the actual malware implementations are commonly less sophisticated, in practice they end up being more effective than their traditional counterparts. Indeed, the lack of proper security measures in combination with barely any human interaction, which consequently means that attacks must be fully automated, results in a high rate of infection.

Based on above mentioned works, we will now discuss what each stage entails, give some examples of techniques used at each stage, and indicate differences with traditional malware where appropriate:

1. Initial Access: Adversaries must first be aware of vulnerable devices in order to attack them. Public IP addresses are continuously being scanned for various exposed and insecure protocols. Telnet and SSH are the most common services being scanned for [MS18; VKH19]. This is likely due to the services providing direct shell access. Aside from tools such as `nmap`[1] and `masscan`[2], the scanner can be a custom implementation. It can also be made part of the malware binary itself. With this approach, the amount of scanned devices, and consequently the number of infections, grows exponential as the malware expands by infecting new devices. The Mirai botnet used this approach to grow quickly in only a few days [Ant+17].

2. Execution: An attempt at executing malicious commands or code can be made after discovering a potentially vulnerable service. First, access must be gained however. This must be automated due to the lack of human interactions with IoT devices. This is in contrast with traditional attacks abusing humans as they are commonly the weak link in security. The two techniques to gain unsolicited access to an IoT device are brute forcing (default) credentials and exploiting Common Vulnerabilities and Exposures (CVE) [Alr+21].
   After having gained access, malware is downloaded and executed. The download can come in various forms such as by FTP, HTTP, or by sending hex or base64 encoded blobs as shell commands. IoT malware employs similar techniques to traditional malware in this regard. However, it relies heavily on shell commands and multiple architectures are supported. The latter is especially important given the variety in Reduced Instruction Set Computer (RISC) architectures.
   While not mentioned by Palo Alto Networks, Inc., the earlier mentioned authors of the honeypot projects, i.e. the authors of [Pa+15], [Dan+19], and [TOS21], note that this stage already includes collection of information and evasion. This comes in the form of shell commands used to test the environment. This is for two reasons. First, the environments between victims may differ. In order for the malware to download and execute properly, the adversary must identify properties such as available tools, writeable directories, and the system architecture. Second, adversaries attempt to evade security testers' sandboxes and honeypots. Testing the environment's believability and halting execution if needed complicates the defenders' jobs.

3. Persistence: Traditionally, persistency between reboots is ensured by writing settings that automatically execute the malware on boot. For example by setting up registry keys or a system service.

[1] `https://nmap.org/`
[2] `https://github.com/robertdavidgraham/masscan/`

IoT malware attempts similar techniques as well. Malware will commonly be ran as the `root` user due to insufficient separation of privileges. This makes persistent installation of the malware easier and even more effective. However, read-only filesystems may complicate this approach. As such, IoT malware leverages the fact that the devices they run on are not frequently rebooted due to lacking human interactions. By taking the watchdog out of the picture, a service that regularly checks whether the system is malfunctioning or overloaded and reboots the device if so, maximum uptime is ensured.
Persistence may also include securing the system. To be more specific, securing it from other adversaries. This ensures that other malware can not compete for resources or even remove the prior one. This is important as IoT devices use constrained hardware. Each infected device is thus of importance. Techniques to lock down the system include changing passwords, halting certain services, or patching the vulnerabilities used by the original malware during its infection. In the latter two cases, a connection must either be kept open or a backdoor must be created. A botnet client connecting outwards is an example of a backdoor.

4. Evasion: Evading detection and complicating auditing extends the lifetime of not only an instance of malware, but the whole malware campaign. Indeed, security researchers can not shut down an operation if they do not have enough information on its infection vector, Command and Control (C&C or C2) server, general working, etc. As such, adversaries targeting IoT devices incorporate evasion techniques in their malware, despite auditing being less prevalent and effective than in traditional hardware due to the constrained hardware, as discussed in section 2.3. Employed techniques are renaming the malware binary on disk or its process to something non-suspicious, such as a common tool, or deleting it completely from the filesystem after execution has started and the binary is completely loaded into main memory. The Hajime botnet incorporates these techniques [EP16]. Furthermore, the creation of logs can be prevented by disabling the firewall and configuring the shell history file to point to the sink `/dev/null`. Evasion can also be applied earlier during the infection process to prevent executing in a fake environment, as mentioned at the end of the Execution stage's description. Techniques to do so are discussed in section 3.1.4. IoT-focused malware also employs anti-analysis techniques, e.g. obfuscation, on a binary level. This helps to evade potential IDSs and firewalls. It also complicates the work of security researchers attempting to study the malware binary. Yet, compared to its traditional counterparts, usage is not as widespread [Alr+21; Tor+21]. Obfuscation techniques include packing the binary into a self-extracting, compressed format with a tool such as `upx`[3], and corrupting parts of the Executable and Linkable Format (ELF) file header that are not used by the OS to execute such files.

5. Collection of Information: Aside from profiling the system in order to assist infection, malware may also collect and exfiltrate other information. The hashed passwords of all users are an easy target. They are stored in the `/etc/shadow` file on Unix systems. This file is only accessible by the `root` user. However, inadequate access control results in adversaries easily gaining access to said user and thus the credentials. Cracking the password hashes is simple given the fact that IoT devices use weak algorithms to save CPU cycles and battery power. Other artifacts can be stolen based on the type of device infected. For example, a Network Attached Storage (NAS) device may contain a large collection of sensitive files. A router, on the other hand, may be used as a Man-In-The-Middle (MITM) for collecting network data.

6. Command and Control: Once installed, most malware clients will periodically contact a server for instructions. This server is called the C&C server. Instructions are not only limited to executing shell commands or attacking a certain host. Management related instructions such as changing C&C server and even self-removal are important as these can be used to stay undetected. That said, Alrawi et al. notes that the usage of C&C servers in IoT malware is not as sophisticated in comparison with desktop malware. IP addresses are hardcoded and employed topologies are much simpler. This might hamper scalability and redundancy [Alr+21]. Communication between client and server can go over custom protocols, or established protocols such as Tor[4] or Peer-to-Peer (P2P) protocols.

7. Lateral Movement: IoT malware is automated due to the lack of human interaction. This means that infections happen via publicly accessible services rather than being initiated from the inside. However, a device may also scan and propagate to the internal network after being infected via the

[3] `https://upx.github.io/`
[4] `https://www.torproject.org/`

Internet. The attack lifecycle for such an attack looks identical to the eight stages being described here, with the small difference of initial access being performed over a private network. The authors of the IoT honeypot projects, mentioned at the start of this section, did not record data on lateral movement. The likely reason for this being the lack of network depth in their setups.

8. Impact: Impact refers to the malware's actions. In [Dan+19] and [Pa+15] it is called Monetization as malware is commonly used to profit the adversary. However, this is not always the case. For example, the Silex malware's sole purpose is to destroy the device it infects [Mic19]. The resource limitations of individual IoT devices are offset by the collective size of the networks they are a part of. As such, possible attacks range from cryptocurrency mining [Mer18], despite the arguable low return on investment, to high volume DDoS attacks. Instructions to perform actions are received from the C&C server.

2.5 Communication of constrained devices

Inter-device communication is a core feature of IoT devices. However, their capabilities, mentioned in section 2.3, also influence the design and choice of communication protocols. Indeed, hardware constraints, such as the usage of a battery, demand small overheads in packet sizes and processing. Furthermore, the physical environment, and placement of devices therein, may impact the reliability and effective bandwidth of communication. This demands Quality of Service (QoS) features to manage the network being lossy while simultaneously minimising latency and managing congestion. Vulnerabilities exist on every layer of the OSI model: in the 4G hardware modules [Shu+19] and microphones [Sug+20], TCP/IP libraries [San+21], the networking protocols ZigBee [Ron+17] and Z-Wave [Bou+20], and application protocols like MQTT [Jia+20; MVQ18] and CoAP [MVQ18][SHB14, Chapter 11]. Given the context of building a honeypot that faces the Internet, physical access for adversaries is impossible. As such, we will now study the high-level workings and features, as well as vulnerabilities, of some popular application level protocols.

2.5.1 MQTT

The messaging protocol MQTT is standardised by OASIS and described in [Ban+19]. The protocol is designed to be easy and fast to implement. This allows for rapid development, as well as low storage and usage overhead, which makes it attractive for IoT projects. MQTT works over TCP/IP. It uses a client-server network architecture and a publish-subscribe messaging model. The model can be compared to publishing and subscribing to a newsletter. The broker, i.e. server, is the distributor while every client is simultaneously a reader and a publisher of messages.

No clients communicate directly. Instead, all communication is done via the broker. Clients may subscribe to topics by means of SUBSCRIBE messages. Any client may also publish application data to topics by sending PUBLISH messages to the broker. The broker relays received published messages to all subscribers of the appropriate topic. In practice, it duplicates and pushes the messages to all clients who subscribed to the topic. Topics follow a hierarchical tree-like structure. Unsurprisingly then, topic semantics are similar to directory paths, i.e. levels are separated by a `/`. When subscribing, wildcards can be used to indicate multiple topics at once.

While the above is enough for a working communication protocol, MQTT includes some additional core features useful in constrained and unreliable environments. First, a Will message can be configured per connected client. This is a message to be published when the client unexpectedly loses connection with the broker. For example, the client may indicate its connection status in a topic. A Will message can make sure this is set to "not connected" in the case of unexpected failure of the client. Further, it is possible to indicate that a message should be retained. The broker must store such messages and publish them to all clients who in the future subscribe to the topic to which the message was published. Last but not least, MQTT includes three levels of QoS: at most once delivery (0), at least once delivery (1), and exactly once delivery (2). In order to manage topics as well as these features, the broker must manage some state. This is done using sessions. When connecting to the broker, a client provides a `ClientID`. This associates a session, possibly from an earlier interaction, with the client. In this regard, `ClientID`s are similar to cookies on the web. Sessions store a client's subscriptions and state of delivery of messages with QoS 1 or 2.

Numerous problems have been found over time in the MQTT protocol itself. Frequently the cause of these vulnerabilities is, sometimes indirectly, the protocol's focus on only requiring a small core set of features to be implemented. By only sparingly using forceful language, e.g. *must* or *required* [Bra97], the specification leaves many implementation details up to the developers. This leads to malpractices and inconsistencies between implementations. For example, the standard states that a receiving entity *may* close the connection if a message containing specific Unicode code points is received [Ban+19, Section 1.5.4]. If a broker decides not to implement this check, it will relay the received message to the subscribing clients without problems. Any receiving client that does implement this check will now disconnect. Combining this with the retain feature, forcing the broker to send the same message to the clients every time they attempt to re-connect, essentially causes a DoS attack against these clients. This is only one of the implementation-based weaknesses discussed by Maggi, Vosseler and Quarta [MVQ18].

In terms of authentication, the standard provides for simple authentication schemes. Even so, despite being recommended, how and whether authentication is performed is left up to the implementation [Ban+19, Section 4.12]. Among other authorization problems, both Jia et al. and Wang et al. describe the well-known problems with `ClientID`s [Jia+20; Wan+21]. Despite serving similar purposes, both the standard and industry do not handle `ClientID`s with the same care as cookies. Instead, they are sometimes shared publicly and/or generated based on predictable patterns. `ClientID`s are also disjunct from credentials. Thus, a client may authenticate with a set of credentials but provide the `ClientID` of another client. This may cause problems as IDs are considered to be unique within a broker. The specification states that, when a client provides a duplicate `ClientID`, the broker must disconnect the client that used said ID up till that moment [Ban+19, Section 3.1.4]. Doing this repetitively is a DoS attack against the original client. Additionally, the session associated with the `ClientID` is not reset, unless requested by the newly connecting client. Essentially, this allows for session hijacking attacks. Last but not least, the standard does not dictate the authorization of topics. Using wildcards, i.e. subscribing to `#`, allows a rogue client to eavesdrop on all traffic.

2.5.2 UPnP & SSDP

The Universal Plug-and-Play (UPnP) protocol was a collaborative effort between vendors to standardise the discovery and control between consumer devices. Nowadays it is being managed by the Open Connectivity Foundation [Fou22]. UPnP was not explicitly designed for the IoT space. It is a "chatty" protocol, which makes it unfavourable for low-powered devices. Yet, the protocol has been widely adopted in the IoT space. This is due to their similar interests of providing convenience to users through seamless interoperability [Kay+20]. UPnP is a Machine-to-Machine (M2M) protocol leveraging zero-configuration networking. Meaning, devices create a network between themselves without human intervention. This encompasses the configuring of IP addresses and host names, as well as the discovery of services on the network [Cor12]. Discovery of other devices is done using an HTTP-like message structure over UDP by means of the Simple Service Discovery Protocol (SSDP), a separate protocol that has been absorbed into UPnP. Data exchange, configuration, and control is done over TCP/IP using the Simple Object Access Protocol (SOAP). SOAP allows for a structured way of exchanging application data using XML over HTTP.

The UPnP specification [Don+15] distinguishes between the terms service, device, and control point. A service logically models (parts of) the state, and features of, a physical device. A device is a logical model grouping multiple services or other devices. These concepts disconnect the physical device from its virtual UPnP representation. Lastly, the control point can be seen as the client. It performs all UPnP related tasks such as discovery, fetching information on other devices, and passing instructions to its own devices and services.

The UPnP protocol follows a distributed M2M approach. A device can advertise its capabilities in two different ways over UDP. First, by multicasting HTTP-like messages using the NOTIFY method to `239.255.255.250:1900`. Alternatively, it can reply to an M-SEARCH message if it is able to provide any of the services requested in said message. These search messages are sent either unicast or multicast. A device's advertisement contains basic information on one of its devices such as UUID, type, and a URI locating the device description for further interactions. The resource at this URI lists additional device information, and most importantly, extensive information on the available services. This service information includes possible actions to get and set data, along with the required parameters and URIs for said actions. UPnP defines XML schemes in order to standardise the structure of the above. Furthermore, URIs for presentation and eventing can be found in the device description. The presentation URI points

to a page showing the same information, and allowing for the same control, as above, but it is designed for human consumption. Eventing, on the other hand, allows a control point to subscribe to changes in the state of another device.

SSDP can be abused to launch reflection attacks. This is the result of combining the concepts of amplification and IP spoofing. First, it is possible to send an M-SEARCH message requesting all capabilities from a device using the `ssdp:all` filter. This will result in a response per available service. In other words, the replying device will amplify the amount of messages being sent, and consequently the used bandwidth. The size of a response also tends to be bigger than that of the corresponding request. The second concept, IP spoofing, allows one to change the source IP address of a message. This is only possible due to UDP being a connectionless protocol. In conclusion, an adversary can send a small number of packets, with the source IP spoofed to be that of their victim, to UPnP enabled devices. These devices essentially reflect packets onto the victim. Unfortunately, misconfigurations expose enough UPnP enabled devices on the Internet such that this attack can be used to generate up to 100+Gbps of bandwidth [Maj17a].

Aside from generic exploits such as command injection [Squ08] and stack overflows [Moo13] due to implementation bugs, the SOAP services of UPnP devices tend to be vulnerable to protocol abuse [Squ08; Kay+20]. The UPnP specification does not impose the implementation of any authentication or authorization. This means that anyone with network access can invoke service actions. While Squire describes this attack for Internet Gateway Devices (IGD) specifically, any UPnP enabled device may include security critical actions. They exemplify this attack with the `AddPortMapping` action that, as the name implies, registers a new port mapping. An attacker may have access to the IGD's SOAP service due to e.g. a firewall misconfiguration and leverage this to expose other internal network services to the outside world. It is also possible to map external resources. This converts the IGD into a proxy for the attacker to leverage in other attacks [Res18]. Lastly, we also mention the possibility of spoofing SSDP advertisements, which in turn enables phishing. This is again due to the lack of access control and verification of advertised services [Kay+20].

2.5.3 CoAP

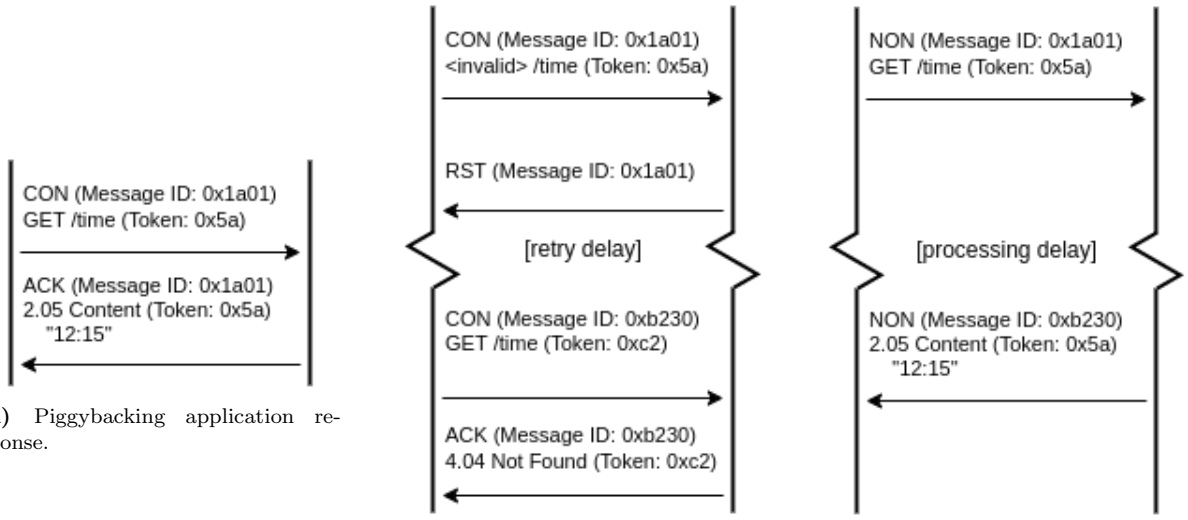
The Constrained Application Protocol (CoAP) is specified in RFC7252 [SHB14]. It is an M2M protocol specifically designed for constrained devices and networks. Similar to UPnP, it supports discovery of services, and a client-server interaction model for control. This client-server interaction resembles that of the web. Furthermore, CoAP follows a RESTful design in order to allow for compatibility with existing web resources, all the while avoiding HTTP's relatively big complexity and packet size[5]. CoAP was designed such that no fragmentation should occur, i.e. a message fits into a single datagram. The statelessness of the protocol allows for the implementation of caches and, due to the similarity with HTTP, cross-protocol proxies. These can further aid to reduce resource usage of the constrained devices. Additionally, CoAP works over UDP as the connectionless protocol uses less resources than TCP. However, optional reliability, i.e. QoS, and encryption are supported. UDP also enables multicast communication. Requesting the path `/.well-known/core` on a device, optionally with a filter, allows a CoAP client to discover the services offered by the device. CoAP is conceptually split into two models: one handles the lack of connections and reliability of UDP (messaging model), and the other manages the web-like provisioning of application resources (request/response model). Figure 2.2 illustrates the difference in models by means of three different scenarios.

The messaging model consists of four possible message types: CON, NON, ACK, and RST. Application data is sent using either CON or NON messages, with the prior demanding confirmation of delivery by means of an ACK message. An exception to this may be made when the application presents the response data in time. This data may then "piggyback" on the ACK message, saving resources by foregoing an additional round trip. This can be seen in fig. 2.2a. Retransmission is attempted if no confirmation is received in time. The retransmission mechanism implements exponential back-off and a maximum amount of attempts as to prevent flooding. An RST message is sent in response to a CON or NON in case of a processing error. The difference with an application error is shown in fig. 2.2b. The first message in this figure is malformed. This results in the CoAP service being unable to process the message, causing it to reply with an RST. The message that follows after the retry delay is properly formatted. However, the requested resource is unavailable, which results in a 4.04 application error. Message IDs are used to

[5] Unlike HTTP/1.x, which uses plain ASCII, HTTP/2 implements compression [BPT15]. This helps to bring HTTP/2's packet sizes closer to those of CoAP.

uniquely identify sent CON and NON messages, and to match corresponding confirmations. Note that responses must be sent with the same QoS as the request they are answering.

Exchanging resources is done using an HTTP-like format. Requests must specify a method and a URI. Optional are a payload, if the method requires it, and options, the equivalents of HTTP headers. Responses contain a status code and optionally media corresponding to the requested resource. The supported status codes are a subset of HTTP, leveraging the existing categories of e.g. 2XX for success and 5XX for a server error, enriched with codes specific to CoAP. The strength of CoAP is that these features are encoded in a more compact way than HTTP, despite their conceptual resemblance. For example, status codes are encoded as a single byte with the first 3 bits specifying the code class and the last 5 bits the specifying the code detail. Tokens are used to identify corresponding requests and responses. They differ from message IDs as can be seen in both fig. 2.2b and fig. 2.2c. In the latter, the message IDs differ as the messages are unrelated on the messaging model level. Indeed, NON messages follow a fire-and-forget approach. Yet, the tokens match as the two messages carry a request for data, and the response containing said data, respectively.



(a) Piggybacking application response.

(b) Message processing errors differ from application level errors.

(c) Message IDs and tokens are parts of different conceptual CoAP models and thus serve different purposes.

Figure 2.2: Three example CoAP interactions attempting to fetch the current time. These exemplify the relations between messaging model and request/response model. Selected headers for both models are shown as the first and second line respectively in each message block.

Packet encryption is supported with IPsec [FK11] or Datagram TLS (DTLS) [RM12]. Encryption can introduce delays and consume high amounts of energy given the computation costs. This has prompted research towards more optimised DTLS implementations [Har+17; MEB16; Cap+15]. Certificates can also be leveraged to implement authentication and authorization. The original RFC does not support DTLS for multicast communication as DTLS handshakes, and consequently connections, are made between unique pairs of entities. This means that in order to emulate encrypted multicast, a CoAP client would have to establish an encrypted session with each server that responds to the multicast message. This is not feasible due to performance reasons. However, it is possible to extend CoAP with the concept of "group keys" in order to support the above [Par20].

The CoAP RFC [SHB14, Section 11] itself explicitly discusses likely security vulnerabilities and possible mitigations. These concerns include implementation errors such as incorrect parsing, amplification and IP spoofing, the limitations of constrained devices such as lacking entropy limiting proper encryption, and MITM attacks resulting from improper proxy usage and implementation. Despite the RFC's warnings, Wang et al. was able to identify production implementations vulnerable to reflection attacks [Wan+21].

2.6 Conclusion

Concluding this chapter, we learned that IoT is a highly heterogeneous concept. The hardware, design, protocols, and goal of each device are vastly different. This leads to the creation of products of which the security is 20 years behind current best practices. Adversaries are able to easily exploit these weaknesses and consequently have significant impact on society. However, not only do the devices lack a secure design, the popular application layer protocols do as well. A security by design approach should thus be required. Additionally, more abstraction layers might aid in both the secure design and implementation of IoT devices.

Chapter 3

Honeypots

One of the first books written on the topic of honeypots is that of Spitzner from 2002 [Spi02]. In 2012, a comprehensive guide on honeypots was published by ENISA [ES12]. And in 2021, Franco et al. surveyed honeypot projects focusing on IoT and CPS [Fra+21]. While far from being all the literature on the topic, these works spanning almost two decades prove that honeypots are a well-established concept in both academia and the industry, and applicable in various areas of technology. All these sources define honeypots similarly. The goal of a honeypot is to incite and capture unauthorized interactions with it. It does this by looking like a real production resource to adversaries. Analysing these interactions can lead to a better understanding of the state of security surrounding the resource imitated by the honeypot. The lack of actual production value means that any interaction is assumed to be suspicious by definition.

Caution must be taken during design and deployment to ensure a honeypot does not interfere with an actual production environment, as interference may cause poisoning of assets. This can happen in both directions. For example, fake data from the honeypot can pollute production data. This can happen either accidentally, or intentionally by an adversary who discovers the honeypot's nature. Production systems may also be infected or even taken over if the honeypot is not properly segregated from the production environment. However, the production environment can also pollute the honeypot. While all interactions with the honeypot are assumed to be malicious, this only applies if the system is implemented correctly. Otherwise, unaware, well-intentioned users may produce honeypot logs. These can hinder analysis.

In order to build one, we first need to understand the components that make up a honeypot as well as its requirements. Only by doing so will the final product be able to perform as intended. Architecturally, a honeypot consists of two parts as defined by Fan et al.: the decoy and the captor [Fan+18b]. These are to attract and handle interactions respectively. The decoy can be any type of resource and is only limited by the honeypot's goal. This in turn is defined by the operator and environment the honeypot has to fit into. The captor, on the other hand, must be hidden as to not alert the intruder. Its responsibilities are securely handling all data served to the intruder, as well as captured for the owner. These two components will now be discussed in further detail, along with associated requirements.

3.1 Decoy

The decoy part of a honeypot is, as the name implies, a resource used to capture the attention of adversaries. This means it must be visible, believable, and desirable to potential adversaries. Fan et al. mentions several specific characteristics of a decoy [Fan+18b]. These can be summarised under the more generally used concepts of role and level of interaction, which we discuss below.

It is important to consider the type of resource, goal, and target audience of the honeypot before building its decoy. This ensures that efforts are appropriately allocated and the resulting product achieves its goal. It is only by hooking and keeping an attacker engaged, that they will share relevant data. For example, a honeypot could be designed to research the current threats by black hat hackers to a certain service. The honeypot system would then require said service to be exposed to the Internet. The service, and possibly the whole system, would have to be highly realistic in order to imitate an actual system as closely as possible. Only as such is capturing accurate data on the currently employed attacks possible.

On the other hand, a honeypot might instead be designed for detecting abuse in a company's network, similar to an IDS but without signature requirements. As the goal would be to determine whether an attack happens, rather than how it happens, simple logging and limited believability of the service would suffice.

3.1.1 Role

The role of the honeypot in an interaction is dictated by the resource it imitates. We can think of this classification as "what is the honeypot?".

A **honeytoken** is a honeypot in the form of a piece of data [Spi03]. It can be a file, a database record, credentials, etc. Despite looking like production data, it should not interfere with, i.e. poison, the actual data. Honeytokens allow detection of unauthorized access, as well as fingerprinting of adversaries. Tokens should be unique to ensure accuracy. Keeping all this in mind, proper generation of tokens is not trivial [Ber+11].

Client-side honeypots emulate client applications, such as browsers. They are an active type of honeypot, purposefully connecting to servers with the intention of evoking an attack from said server [SWK+07].

In contrast to client-side ones, **server-side honeypots** are passive systems. They emulate and expose services to be abused. The goal of server-side honeypots is to capture automated or manual scans and probes, as well as post-intrusion interactions [Naw+16]. The placement of server-side honeypots dictates their use case, as explained by Spitzner [Spi02, Section 12.4]. Honeypots in the DMZ might deter adversaries as it wastes their time. Additionally, given that honeypots look like production systems, placing them in the DMZ allows one to study how an infection/intrusion looks. This knowledge can be used to search for abuse of actual production systems. Similarly, a honeypot on the internal network can be used to detect infections from within, e.g. coming from a malicious e-mail attachment. This internal placement also has the advantage of being able to detect insider threats, such as a rogue employee. Research honeypots are less restrictive for an attacker as to allow more detailed information gathering. However, this puts production systems at risk. Meaning, the two should be separated as much as possible.

3.1.2 Level of interaction/fidelity

A honeypot is meant to imitate a production resource. Its goal dictates the extent to which the implementation of the decoy must look and feel real to adversaries. In other words, any resources spent on an interaction with an adversary, after acquiring the desired information from them, is wasted. However, insufficient fidelity may also adversely affect the honeypot's effectiveness. A system that is discovered to be a honeypot loses its value. Adversaries may prematurely end interactions, block list its IP address, or even poison it with false data. The efforts spent on the implementation result in differences in fidelity and possible levels of interaction [Spi02, Chapter 5][Fra+21; Fan+18b]. We can think of this classification as "what does the honeypot do?".

In their book *Honeypots: tracking hackers*, Spitzner defines a low and a high level of interaction [Spi02, Chapter 5]. Due to being on a continuous scale, a medium level theoretically exists and is sometimes used in literature. However, a more interesting third option exists that combines both low and high level components, that is the hybrid interaction honeypot. The differences between levels of interaction fall into following categories: cost and complexity of implementation, risk/impact of abuse, fidelity of environment and recorded data, and lastly scalability and maintenance [Fra+21]. While measuring these characteristics objectively and discretely is difficult, the relative relations can be represented as in fig. 3.1. Note that these characteristics are heavily codependent in practice. For example, consider a project with high cost and theoretical infinite scalability. In practice, all this scalability will be for naught as it is limited by to the costs. Deducing these characteristics from existing honeypots is a trivial academic exercise. However, the inverse process, i.e. determining the resulting level of interaction given the desired levels of characteristics, is interesting in practice. Exploring the goals of a honeypot project enables a more efficient and streamlined development process, and consequently results in a more appropriate product. We will now go a bit more in depth on the three levels of interaction using the descriptions provided in [Spi02, Chapter 5], [Fra+21], and [Fan+18b].

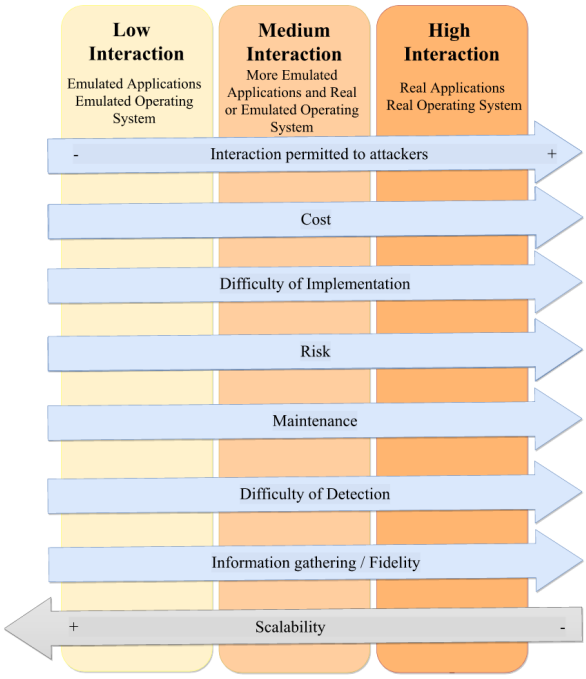


Figure 3.1: The characteristics impacted by the level of interaction of a honeypot can be placed on a continuous scale. The arrows indicate the direction of "increase". For example, the cost of a low interaction honeypot is low while its scalability is high. Figure taken from [Fra+21].

A **low interaction** honeypot only provides limited functionality compared to the resource it simulates, while attempting to look as similar as possible at first sight. The implementation uses the original service or re-implements a bare bones version of it, allowing it to be set up easily and quickly. Due to being small and simple, maintenance and scaling is easy. There is also little risk that an adversary misuses the system, given the lack of features. However, the lack of features limits the possible interactions and increases the possibility of detection, consequently limiting the captured data. These honeypots work well to detect simple contextual data such as meta data and initial payloads.

High interaction honeypots are on the other end of the spectrum. They are (nearly) fully authentic emulations of a system, providing a full OS and file system for adversaries to interact with. It is possible to adapt an existing system, but this does not make it easier to build the honeypot. In contrary, due to adversaries being given access to a full system, the risk of abuse is high. In other words, the honeypot might become complicit in illegal activities. Proper management, such as the configuration of firewall rules, bandwidth limitations, and access control, is essential. Honeypots with a high interaction level are very effective at capturing end-to-end attacks and logging them in detail.

Lastly, **hybrid interaction** honeypots exist. These are setups combining low and high interaction honeypots, effectively providing the benefits of both techniques [ES12, p.19-20]. Low interaction honeypots are deployed lavishly as endpoints to increase the reach of the system. These decoys perform an initial assessment of the intruder. If deemed appropriate, intruders are passed on to high interaction honeypots as to acquire higher fidelity data. The handover happens transparently as to not alert the intruder. It is also possible to pre-configure the high interaction honeypot with earlier captured data. This increases the accuracy of the honeypot.

3.1.3 Visibility

Naturally, adversaries must be aware of the existence of a honeypot's hosted services in order for interactions with it to happen. Discovery of the honeypot can happen similarly to the initial access stage mentioned before in section 2.4.2. That is, exposed services can be found by scanning public IP addresses using automated tools. Nevertheless, automation hampers flexibility of attacks. Malware that propagates by means of self-replication will scan and attempt to intrude following a certain set of rules it was programmed to do. Humans, on the other hand, understand context. This allows for more tailor-made

interactions. The target may also contain artifacts of interest, such as documents containing confidential information, that humans have an easier time interacting with [TOS21; BN17]. Differences in experience and background may also affect a human's actions. Post-intrusion behaviour of humans and bots is thus different. Although exposing a honeypot's decoy publicly on the Internet is simple and certain to yield results, directly "advertising" it to humans should be considered by its operators.

It is important to note that the advertisement's goal is attracting adversaries. It is thus a part of the honeypot's decoy and must thus have the same properties of being believable, desirable, and visible. In practice this means that the advertisement's story must be appropriated to fit both the context of the honeypot, as well as that of the distribution medium. It must also reach as wide an audience as possible. Going a step further, the advertisement may be made to contain honeytokens, which enables the fingerprinting of adversaries.

In order to reach possible adversaries, advertisements should be made in locations that they normally visit. The most popular avenue for advertising honeypots is Pastebin-like web sites [Mis+18; Fra+18a; BN17; LOS16; Hil+20; OMS16]. It is not uncommon for adversaries to share actual (samples of) stolen data on these web sites. These sites are also being crawled with the purpose of discovering real credentials in e.g. config files posted by security-unaware users. The authors of both [Mis+18] and [BN17] created bots that posted periodically to several Pastebin-like web sites. This increased their chances of being noticed. The importance of visibility is made clear by Hilt et al., whom only made two posts and reported no interactions based on them [Hil+20]. A similar outlet is hacker forums used to trade in leaked information [Fra+18a; BN17; OMS16]. Researchers mostly craft similar stories when posting on these sites. They pretend to be in possession of a great amount of credentials and offer to share a small sample. However, the sample contains only honeytokens for the researcher's honeypots. Sometimes dark web alternatives to both Pastebin-like web sites and hacker forums are also used.

Taking a step back from pretending to be sharing leaked information, some researchers create leaks themselves. For example, services such as Google Docs® are a popular way, for businesses and individuals alike, to share documents online. However, faulty configuration of access control policies can lead to the disclosure of their contents [LOS16]. Thus, a way to advertise is by creating a document containing a fake story and honeytokens, and then sharing it publicly. Another example is purposefully installing malware, known to contain a keylogger, in a Virtual Machine (VM) and then interacting with it [OMS16; Aki+18]. This can be a very effective approach as it is known that the information will reach adequately skilled adversaries. Nevertheless, this is also the most laborious technique as it requires the acquisition of appropriate malware and proper handling of the malware in order to prevent accidental infections.

3.1.4 Believability

As mentioned in section 3.1.2, a honeypot that is detected as such loses all its value. Honeypots differ from real resources at some level, be it by being a complete re-implementation or simply due to the presence of instrumentation. These differences can be detected by fingerprinting techniques. Some examples are simple banner-based detection [Mor+19] or a more advanced detection of discrepancies in re-implementations of protocols [VC18]. Furthermore, the complete lack of some tools or commands can be detected. For example, Mirai variants echo binaries to standard output at the initial stages of infection. The system architecture can then be determined based on the strings within said binaries. This information is needed such that the correct malware sample can be downloaded. Yet, the attack may be cancelled prematurely if the honeypot does not implement the correct behaviour [mor17]. Similarly, the command `busybox <random_string>` is commonly used for the detection of discrepancies in behaviour of commands. BusyBox[1] is a customisable collection of common Unix tools compiled into a single static binary. It is frequently used in IoT firmware due to its portability and small size. However, the invocation of a non-existing command may not be handled properly by honeypots providing re-implementations of tools, instead of the original tools themselves [Pa+15]. While seemingly naive, these techniques are effective at detecting numerous low to medium interaction open source honeypot projects. Although each instance of these issues could individually be fixed with little engineering effort, fixing each occurrence in order to near perfectly imitate the original resource can quickly become an impossible task. Questions regarding the intended interaction level of the honeypot should arise at this point. Researchers using off-the-shelve honeypot solutions are recommended to make the system as unique as possible while simultaneously making it blend in with the environment in order to thwart fingerprinting. Examples of

[1] `https://www.busybox.net/`

such tweaks are configuring which services are exposed, customising the presented banner and hostname, and matching the OS used by surrounding production systems.

On the other hand, high interaction honeypots tend to use parts of an actual system as a basis. This foregoes or complicates some of the aforementioned fingerprinting techniques. Unfortunately, many other methods exist. These methods attempt to detect the virtualisation software, e.g. emulator or VM, in which the malware is being executed. Commonly, this is done such that the binary refuses to run in a malware researcher's sandbox. However, nowadays this approach is not as applicable for malware targeting servers and PCs, as virtualisation is highly prevalent due to e.g. cloud computing. For IoT malware, on the other hand, these techniques are still useful due to the lack of effective emulation environments. For example, virtual peripherals exposed to the guest OS contain identifiable properties [SM15]. Timing attacks are another possible technique. Virtual environments are slower than bare metal devices due to the added layer. Recording the duration of a (set of) instructions and statistically comparing the results with control devices could reveal that the presented system actually runs in an emulator or VM [HR05; SM15]. The difference in performance between virtualised and bare metal grows smaller each day as the software is optimised. However, even these optimisations can be leveraged to perform timing attacks. Caching is a simple, yet effective, technique to improve performance. As intended, it changes the expected amount of CPU cycles required for an operation. The unfortunate side effect is that this difference can now be used for fingerprinting [Jan+19; Gar+07]. Sometimes, Easter eggs are even added to CPUs that are not known or overlooked by the developers of the virtual counterparts. The presence or absence of this Easter egg is then an effective indicator to the nature of the machine [Fer07]. Kedrowitsch et al. presented the idea of using Linux containers to remove artifacts related to virtualisation, and to improve performance [Ked+17]. Nonetheless, Miramirkhani et al. observes that even when all the above virtualisation-related artifacts are gone, the artificial environment can still be fingerprinted using, what they call, wear-and-tear artifacts. The idea is simple to understand: usage of a real device creates artifacts such as temporary files on the filesystem. However, honeypots are deployed in a clean state and frequently reset in order to keep it that way. Looking for these wear-and-tear artifacts is simple and their presence or absence says a lot about the device [Mir+17]. Similar to the discussion on lower levels of interaction, attempts at preventing detection can be made. Some may be as simple as matching the VM's resources to that of the original hardware or masking virtual peripherals' properties. However, others require huge engineering efforts such as making the emulator time-accurate [CNZ20].

3.2 Captor

Aside from the decoy, the implementation seen by adversaries, several considerations have to be made concerning the implementation of the captor. The captor makes up everything behind the scenes, such as logging, access control, and handling of responses. In other words, it is the security instrumentation that converts an everyday resource into a honeypot. Proper implementation of the captor lowers the risk of abuse of the honeypot. It also ensures that captured data is enough, verbose, and properly structured. Similar to the discussion on decoys, Fan et al. goes in depth on specific characteristics of captors [Fan+18b]. Again these can be traced back to more general requirements, this time defined by the Honeynet Project [Pro04; Pro06].

3.2.1 Data control

Data control pertains to the control of an adversary's actions. The main purpose of honeypots is aiding in the (research towards) protection of production assets. Limiting the associated risk of compromise and abuse, especially for high interaction honeypots, is thus of foremost importance, even above capturing actual data. However, note that limiting an adversary increases the probability of detection.

Control of both ingress and egress traffic must be implemented. This allows proper control of adversaries' actions. Ingress traffic can be monitored to generate intrusion alerts. Ingress traffic can also be blocked. This allows the filtering of Internet scanning services such as Shodan from the logs [Dan+19] or of adversaries whom discovered the honeypot's nature and attempt to poison the data. On the other hand, egress traffic is harder to manage. For example, the honeypot could fall into the hands of an adversary and join a spam e-mail botnet. In this scenario, blocking all traffic going out to port 25 (SMTP) would be effective at preventing abuse [ES12, p.29-30]. While the honeypot's resources would still be abused, no harm would be done to other Internet users. However, blocking by port or protocol is not an effective approach in general. Adversaries may craft their own protocols on top of TCP or UDP, or use

unconventional ports to communicate with e.g. their C&C server. The opposite is also true. Blocking commonly used ports, e.g. port 80, to prevent adversaries from downloading their malware binaries could potentially disrupt the normal workings of the emulated firmware. As such, data control by means of bandwidth limitations is preferred. Using multiple control services is recommended to ensure redundancy. In case of failure, rules should fall back to the closed state.

3.2.2 Data capture

To know what actions adversaries performed, their actions must be logged. Captured data has to be structured, clean, and complete. This makes sure a fruitful analysis of it is possible.

To ensure that a complete picture of a situation can be formed after the fact, enough logging has to be implemented. An attack must be logged from the start till the end, along with possible branches. Capturing multiple features, e.g. network frames and shell interactions, allows for analysis from multiple angles. Too much logging is theoretically not possible. Data can be filtered if unneeded, but it can not be captured after an incident has already happened [Spi02, Section 12.5.1]. For the same reason, data should be stored in its raw form. Processing is namely a form of filtering. However, adherence to a standardised format, including for time stamps, facilitates analysis. Honeypots should not interfere with the production environment. This is not only to prevent complications for the environment, but it also prevents poisoning of captured data. Similarly, storing logs on a separate (remote) device protects its integrity from changes by adversaries.

3.2.3 Data collection

Data collection pertains to the aggregation of captured data from multiple distributed honeypots. Such a distributed network can be used to e.g. analyse geographic correlations in attack campaigns [Vas+15]. The difficulty of distributed setups is to structure and synchronise data captures. For example, synchronising time stamps and using a structured naming scheme for nodes [ES12, p.30]. This is again to ensure that proper analysis is possible.

3.3 IoT service honeypots

In 2012, an anonymous researcher created a distributed botnet that was able to scan the whole IPv4 range in a single night. Their goal was to study the amount of devices exposing Telnet with default credentials to the Internet. Results showed that there were "several hundred thousand unprotected devices on the Internet" [ano12]. Despite the huge ethical debate surrounding it, the research raised awareness of the extent of the problem. Starting around 2014, both academics and adversaries picked up interest in the network level security of embedded devices, as can be seen in the boom of research projects [Fra+21; Fas+21] and scanning frequency of e.g. Telnet services [Pa+15] respectively. Mainly low and medium interaction honeypots were initially created to understand the threats.

Projects such as cowrie[2] and MTPot[3] provide a low interaction Telnet shell simulation. These allow capturing shell interaction from adversaries. The shell environment is incomplete and differs from a real one as it is simply a re-implementing of certain commands. These discrepancies can be detected, alarming adversaries and halting interaction prematurely [mor17; Cab+19].

Pa et al. built a hybrid honeypot called IoTPOT [Pa+15]. Adversaries can interact with a "dumb" but learning Telnet shell simulation. Login banners are sourced by scanning other devices. Meanwhile, responses to shell commands are learned by running the commands in an OpenWrt[4] emulation. OpenWrt is an open source, modular, and light weight Linux-based firmware for embedded devices. This solves the issues seen in cowrie and MTPot. Using a real OS as backend improves fidelity and decreases the likelihood of being detected, allowing for more in-depth interactions and analysis. Multiple OS emulations were used to support different CPU architectures. The captured data was then used to study the general flow of Telnet based attacks, contributing to the discussion in section 2.4.2. Similarly, Falcom [Fra+18b] implements a high interaction honeypot giving adversaries full access to a 32-bit MIPS OpenWrt emulation. As opposed to IoTPOT, however, the Telnet service provides direct shell access to the OS.

[2] https://github.com/cowrie/cowrie/
[3] https://github.com/Cymmetria/MTPot/
[4] https://openwrt.org/

Despite Telnet being the most popular service being scanned for [MS18], other services are not being neglected. Vulnerabilities in e.g. UPnP are not new [Squ08]. U-PoT [Hak+18] is a framework that crawls an existing device's UPnP space. It does this in an attempt to collect all endpoints, and learn associated actions and how they affect the state of the device. This data can then be used to generate a high fidelity, full software simulation of the device's UPnP service. Tabari, Ou and Singhal simulated an IP camera using a low interaction web server based on D-Link® devices, and Telnet servers [TOS21]. They observe that while most traffic targets Telnet, this traffic simply brute forces credentials and is done by bots. Meanwhile, attacks on web servers tend to abuse CVEs and are done more manually. ThingPot [WSK18] goes a step further. Instead of focusing on an individual device, it simulates a network of IoT devices communicating using the Extensible Messaging and Presence Protocol (XMPP). The smart devices' hub is accessible via an HTTP REST API and an XMPP client. Captured data showed highly device-specific interest in the REST API, while the XMPP path was mostly ignored. The lack of interest in XMPP is likely due to the inexperience of adversaries with the protocol.

3.4 Firmware re-hosting

Notice how the honeypot projects mentioned in section 3.3 tend to only cover one or a few IoT services and characteristics each. The closest they come to running an actual IoT device, combining multiple services in a way that is production ready, is the usage of OpenWrt by IoTPOT and Falcom. OpenWrt is simple to run on arbitrary hardware as it is designed to be as generic, and thus compatible, as possible. In contrast, using consumer firmware in research is not easy due to the limitations incurred by the hardware, e.g. cost, scalability, and difficulty to use traditional instrumentation. The tight coupling between hardware and firmware, such as interrupt driven code and lack of hardware abstraction layers, as well as the heterogeneous nature of IoT devices, e.g. the variety of architectures and possibly (undocumented) custom instruction sets, complicates the emulation of arbitrary firmwares [Spe+21]. This is a major stumbling block for the creation of high interaction honeypots. Indeed, out of the 37 surveyed projects they studied, Franco et al. identified only 4 that were high fidelity, full device emulations of IoT devices. Half of these required an actual device as backend [Fra+21].

We will now explore several projects attempting to re-host IoT devices. Re-hosting is the practice of virtualising (parts of) a device. Note that these are not all honeypot projects, but projects aiming to emulate devices with the goal of dynamically analysing them first and foremost. That is, reviewing software by means of executing it. This is in contrast with static analysis, which entails abstractly reasoning over software's workings [Ern03]. Nonetheless, an interactive emulation can be turned into a honeypot. A classification of firmware re-hosting techniques already exists, namely that of Fasano et al. [Fas+21]. They identify four approaches: hardware-in-the-loop, symbolic modelling of peripherals, a hybrid of hardware-in-the-loop combined with symbolic modelling of peripherals, and pure emulation. However, this classification was created with dynamic analysis in mind. As a result, not all of their described approaches are suitable for interactive emulation, and consequently suitable to be turned into a honeypot. For example, symbolic modelling is a technique that explores every possible path of execution of a firmware, but does not allow interactivity. Due to the difference in focus, their classification also does not discuss all properties relevant to building a honeypot with said techniques. As such, we create our own classification that focuses on the extent of virtualisation. This classification is created by identifying overarching ideas between several firmware re-hosting projects. The general ideas, strengths, and weaknesses of each of the approaches are discussed by means of comparing them to each other.

3.4.1 Full device proxy

This approach builds a system in which the whole interactive environment is a physical device, and software is only used as instrumentation. Practically speaking, adversaries get to interact with real devices, both hardware and firmware, through proxies. Chameleon [Zho19], IoTCandyJar [Luo+17], and SIPHON [Gua+17] are examples of full device proxy honeypots. Both Chameleon and IoTCandyJar consist of three major components: responder, evaluator, and scanner. The responder receives requests and attempts to respond appropriately from cache. In the case that the responder has no fitting answer, the scanner will search for a physical device to learn the response from. The evaluator's job is to scan requests for security threats. If it finds a request to be malicious, the evaluator will prevent the relaying of said request to physical devices in order to protect them. This architecture results in high fidelity and flexibility, allowing the setups to imitate hundreds of different devices. Machine learning is

employed in the case of IoTCandyJar to further improve fidelity, increasing the duration of interaction with adversaries.

These setups sound quite promising as they manage to create a flexible, scalable, and high fidelity setup for a relatively low amount of effort. However, attacks requiring multiple consecutive malicious requests can not be fully observed as the evaluator prevents appropriate responses from being sent at the start of the chain of requests. Additionally, access to physical devices is required. The authors of IoTCandyJar leveraged public devices on the Internet. This approach requires ethical consideration when performed on a large scale. Especially considering the possibility of relaying malicious requests if the evaluator fails to detect them. SIPHON, on the other hand, tunnels traffic from public IPs to only a handful of self-owned devices. This foregoes the ethical as well as some technical concerns, in exchange for less flexibility in terms of devices being served.

3.4.2 Peripheral forwarding

The drawback of a full device proxy is that the device is essentially a black box. In order to perform e.g. fuzzing, a security testing technique that provides random input and monitors program execution for errors [SGA07], a controlled environment with the necessary tooling, logging, and enough system resources is required. Emulation can provide this environment. However, without access to peripherals, most if not all emulated software will simply not function. For example, configuration options are usually stored in Non-Volatile RAM (NVRAM). Attempting to read such option from the non-existing NVRAM hardware causes problems for the emulation.

Avatar [Zad+14] and PROSPECT [KPK14] attempt to solve the issue of missing hardware in emulated environments using peripheral forwarding. This approach combines virtualising (parts of) the firmware and using real hardware. Hardware access requests made by the emulated software are forwarded to the actual hardware, and resulting data is fed back into the emulation. Evidently, this requires instrumentation for at least the following:

- Interacting with emulation: Capture hardware requests from the emulation, and replay to it the actual hardware's response.
- Interacting with hardware: Replay hardware requests from the emulation to the hardware, and capture its response.
- Communication between the above.

Avatar and PROSPECT differ mainly in their way of capturing hardware requests. Avatar interfaces directly with the used emulator, intercepting events on a high level. This includes intercepting hardware access. Hooking into events allows for flexibility and extensibility for the researcher using the Avatar framework. However, the authors mention that the framework introduces severe latency, despite optimisation efforts.

Two types of device drivers exist in Unix: character and block devices. The main difference between them is that a character device, as opposed to a block device, does not buffer data [K J96]. Peripherals working with continuous data are generally interfaced with through character device drivers, while the latter are used for storage devices. Looking at Type 1 systems, as defined in section 2.2, the authors of PROSPECT make the observation that peripherals and their (proprietary) character device drivers differ greatly from one device to the other. Yet, hardware interactions must eventually go through the kernel and thus use its standardised API. In order to capture hardware requests, i.e. system calls, virtual character devices were created to be used in the emulated environment. Custom handling of system calls was achieved by means of re-implementing the most prominent ones. PROSPECT's authors state that their implementation results in an insignificant latency for its intended goal of step-based debugging. However, it might not work consistently as not all system calls were re-implemented.

Note that this approach requires dropping and executing a binary on the actual device, which might not always be possible given that best practices dictate disabling or removing debug ports on embedded devices. For the same reason, obtaining the firmware is another challenge. Last but not least, we mention Avatar^2 [Mue+18b], the spiritual successor to Avatar. It is a generalised orchestration framework for hardware, dynamic analysis tools, and emulation tools. Its goal is to facilitate the building of systems using peripheral forwarding.

3.4.3 Virtual peripheral modelling

Peripheral forwarding still suffers from a major drawback. Namely, it incorporates hardware in the design. Cost and scalability are directly impacted by the available hardware. Virtual peripheral modelling attempts to solve this, as the name implies, by means of creating virtual peripherals functionally equivalent to real ones, up to a certain degree. These can be attached to emulators, enabling emulated firmware to boot as if on native hardware. The advantage of such virtual model is that it can be shared with the community once created. Despite this, while manually creating virtual peripherals is possible, missing documentation requires extensive reverse engineering efforts. This amount of effort for a single device is usually not justified given the high variety in hardware.

PRETENDER [Gus+19], Conware [Spe+21], Jetset [Joh+21], and µEmu [Zho+21] attempt to solve this predicament using automated peripheral modelling. PRETENDER and Conware leverage the Avatar^2 framework to build a peripheral forwarding system, allowing them to capture interactions with Memory Mapped I/O (MMIO) and the state of it, as well as hardware interrupts. These traces are then processed to create state machines representing the peripherals, extended with heuristics to allow for interactions unseen in the original traces. The authors themselves note two major pain points: modelling and handling of interrupts and the complete lack of modelling of port-mapped peripherals. Additionally, although these projects' approach is indeed similar to the peripheral forwarding approach and thus comes with similar downsides, the hardware is only required in the data capturing phase. This removes the scalability issue for end users.

To completely remove the necessity of hardware, Jetset and µEmu employ symbolic execution to model peripherals. Symbolic execution was first described by King [Kin76]. It is unlike "traditional" execution, i.e. concrete execution, which replaces variables with concrete values. When encountering a conditional test, concrete values allow exploring only one of two possible branches. An instance of concrete execution is therefore limited to a single path through the application, determined by the actual values. On the other hand, symbolic execution replaces variables with symbolic values. These can be seen as infinite sets of possible discrete values. Following a conditional branch during symbolic execution places a constraint, in accordance with each checked condition, on the corresponding symbolic value. Accordingly, the state of an application under symbolic execution consists of a mapping between application variables and symbolic values, as well as the path constraint. The path constraint is a boolean expression containing all symbolic values and constraints they have to abide by to arrive at the current path, and thus stage, of execution. Theoretically, symbolic execution would allow simultaneous exploration of all application paths. Practically, however, not all paths can be explored as most applications contain an infinite amount of paths, and paths that would not terminate within a human lifespan. Of course, ignoring these paths requires crude heuristics as the halting problem prevents determining the nature of a path a priori.

The authors of both Jetset and µEmu present following insight: firmware code implicitly encodes the expected behaviour of peripherals. For example, if a peripheral is expected to set a specific flag in memory to indicate successful initialisation, this flag will be checked by the firmware. Conversely, faulty data causes the firmware to raise an exception, stall execution, or even completely reset the system. This insight is in line with the workings of symbolic execution. The technique can be applied to determine the correct constraints on data to be returned by peripherals at a given state of execution. To guide symbolic execution down a correct path, i.e. a path validly interacting with peripherals, errors are a simple heuristic to use.

Being able to deduce the workings of peripherals, without requiring actual hardware, sounds promising for the modelling of virtual peripherals. Another upside is that these projects support all three types of embedded systems, discussed in section 2.2. However, unlike symbolic execution, which is an old technique introduced in 1976, modelling peripherals with it is a new idea at the time of writing. While the resulting models work as advertised, several conditions apply as discussed by the authors themselves. First, Jetset requires researchers to input the memory layout, and entry and goal addresses, along with the firmware. This requires some manual effort and knowledge of the target device. µEmu attempts to automatically determine memory layout, but improvements are to be made. Second, similarly to PRETENDER and Conware, only MMIO and interrupts are handled. Third, the resulting virtual peripherals return conditionally valid data. This does not mean it is correct data, i.e. data that makes sense. For example, symbolic execution might recognise that a timer linearly increases its value. Yet, it might not recognise that this happens every second. Instead, it will increase the value on every read. Last but not least, the created models' accuracy steeply drops when passing the point up to which it was trained. Jetset's implementation simply repeats the last known read or write interaction. µEmu, on the other

hand, is able to dynamically switch between the emulation and symbolic execution stage. The authors note that the symbolic execution stage takes around 2 minutes, with the worst case taking up to 10 minutes. This is clearly not usable for interactive sessions for humans.

3.4.4 Full system re-hosting

Last but not least, full system re-hosting attempts to provide an interactive, full-software emulation of given firmware. Whereas the previous three approaches attempt to fit the (virtual) peripherals to the emulated firmware in some manner, this approach makes adjustments to the firmware to achieve a functional emulation thereof. Full system re-hosting focuses solely on emulating Type 1 devices, which use a generic OS as discussed in section 2.2. The generality of the, in practice, Unix-like OSes is what is being leveraged by this approach. Although this limits the amount of firmwares that can be emulated, the abstractions of the OS allow for easier instrumentation. Concretely this means easier handling of missing peripherals, and researchers being able to use traditional tooling. This in turn results in a higher rate of successful emulations, and frameworks allowing for a simple automated setup given only a firmware blob. Using the original firmware's filesystem prevents issues with configuration or missing (proprietary) files and binaries. Although the resulting environments look and feel the same as the original, the high fidelity is only superficial due to building on abstractions [Wri+21]. Instead of attempting to perfectly emulate hardware, high-level interventions patch problems allowing the firmware to run. Kim et al. therefore refers to this technique as arbitrated emulation [Kim+20]. Projects following this approach are Costin, Zarras and Francillon's [CZF15], Firmadyne [Che+16], Honware [VC19], and FirmAE [Kim+20]. Each follows roughly the same major steps while improving upon complications encountered by the previous. The major steps are as follows:

1. Unpacking firmware: Emulators expect a filesystem and a kernel image to run. Unpacking arbitrary firmware blobs provided by vendors is thus the first step. This step sounds deceptively easy considering the existence of powerful tools such as binwalk[5], the de facto open source firmware analysis tool. However, vendor provided blobs may use a proprietary or, by tooling, insufficiently supported format, contain only the changes to be made in order to update the firmware, pack multiple architectures into a single archive, or are structured in a non-standardised manner. These complicate the process of automatically unpacking a firmware into kernel and filesystem.

2. Identifying architecture and entrypoint: The Instruction Set Architecture (ISA), its version, and endianness must be known in order to configure the emulated hardware. The straightforward approach is to read ELF headers from firmware binaries. An alternative is to statistically determine the most plausible architecture by analysing opcodes, strings, or other signatures in said binaries [Wri+21]. As for the entrypoint, this is the binary executed by the kernel during boot, i.e. the `init` process. The difficulty is not only to identify the binary in question in the extracted filesystem as different vendors use different file names and paths, but also the parameters used while invoking said binaries. Techniques to find these range from naive find operations based on filename, to looking for strings in the kernel binary.

3. Post-processing firmware: Modifications to both the filesystem and kernel are made to facilitate emulation. This step includes the most interventions made to facilitate emulation. All mentioned projects replace the kernel included in the firmware with their own. This has several advantages such as consistency between emulations, and decoupling software from hardware as vendors provide customised kernels and kernel modules. It also allows customisations such as logging of system calls, changing environment variables passed to the entrypoint, changing the entrypoint itself, and the intercepting and custom handling of signals. Filesystem changes are e.g. the creation of proper symbolic links from text files representing these links, adding a custom `libnvram.so` implementation, and replacing/adding binaries such as BusyBox. All these changes allow for customisation and instrumentation of the emulation. Of course, while only one kernel per architecture is required, maintaining them is a lot of work. The majority of firmwares require older kernel versions due to compatibility reasons with the included services. Yet, emulators require features from newer kernel versions. The solution, although labour intensive, is to backport said required features.
   The custom NVRAM implementation deserves some additional explanation as it is one of the most important, yet imperfect, interventions made to enable emulation. Embedded devices use NVRAM to store configuration data that is to persist between reboots. This storage is commonly abstracted

[5] `https://github.com/ReFirmLabs/binwalk/`

as a key-value store and accessed via a library. These key-values can be stored on e.g. the filesystem by overriding the original library's functions with custom implementations. This approach is plagued by several problems. First, only roughly half of all firmwares use a library to access NVRAM [Che+16]. Second, returning `NULL` or an empty string as default value can easily cause errors in the emulation process. This is partially solved by some firmwares that include defaults in text files, or hardcoded in binaries. Last but not least, the function declarations might differ between vendors.

4. Preparatory iterative emulation: All but Costin, Zarras and Francillon's project use an preparatory emulation phase. Firmwares perform numerous setup steps during the boot process, such as configuring network interfaces with MACs and VLAN tags, mounting additional filesystems, and generating configuration files. Of course, errors can also occur. The customised kernel captures these by logging system calls. This information is then processed to automatically configure the emulation environment. Some setup steps may be (indirectly) dependant on each other. For example, the web server may not even attempt to start before a working network connection is set up. Yet, the web server contains some important NVRAM keys that are only accessed at runtime. These keys can thus not be registered with the custom NVRAM implementation during the initial emulation. It is for this reason that the preparatory learning phase is iterative.

5. Emulation: Using the results of the above steps, emulation of the firmware is performed. The emulator of choice is QEMU[6]. QEMU is the de facto, open source machine emulator. Success of emulation is tested by checking for working networking by sending an ICMP ping and, if applicable, performing a GET request to the web server.

3.4.5 Evaluation

In this section, we evaluate the applicability of each technique in building a high interaction honeypot using consumer firmware. To recap, table 3.1 lists all discussed firmware re-hosting techniques and related projects. High interaction honeypots are expected to be hard to manage in exchange for higher fidelity, as is implied by their characteristics seen in section 3.1.2. Yet, some limits must exist in order for the honeypot project to be usable and viable in practice. This leads us to comparatively evaluate these characteristics per technique first. Note that honeypot operators are assumed to have permission to use all involved devices. As a result, the amount of available hardware units in this discussion is limited to roughly a dozen. The results of the comparison are shown in table 3.2. After the comparison, we will argue the usefulness of each firmware re-hosting technique. The goal is to determine the technique best fit for a user friendly, high fidelity honeypot.

Re-hosting approach	Projects
Full device proxy	IoTCandyJar [Luo+17], SIPHON [Gua+17], Chameleon [Zho19]
Peripheral forwarding	Avatar [Zad+14], PROSPECT [KPK14]
Virt. peripheral (forwarding)	PRETENDER [Gus+19], Conware [Spe+21]
Virt. peripheral (symbolic exec.)	Jetset [Joh+21], µEmu [Zho+21]
Full system re-hosting	Costin, Zarras and Francillon's [CZF15], Firmadyne [Che+16], Honware [VC19], FirmAE [Kim+20]

Table 3.1: Listing of all firmware re-hosting projects, per re-hosting approach.

- Cost: The financial cost of running a honeypot heavily depends on the amount and the computing power of hardware involved, and the necessary maintenance. Both Full Device Proxy (FDP) and Peripheral Forwarding (PF) require actual IoT devices for the running environment. Additionally, gear and work hours must be dedicated to studying and hooking them up to instrumentation, e.g. find a debug port and communicate with it properly. Virtual Peripheral Modelling (VPM) projects PRETENDER and Conware have the same constraints during the training phase. Jetset and µEmu, on the other hand, require computing power relative to the complexity of the firmware due to the usage of symbolic execution. Although emulation in general requires *some* resources from the host computer, the virtual peripheral models are simple enough so that their impact is negligible. Emulation of firmware itself is also relatively cheap. This is due to the constrained

[6] `https://www.qemu.org/`

hardware used by IoT devices. As such, both the usage of virtual peripherals and Full System Re-hosting (FSR) have a low cost.

- Complexity: Some complexity is involved in the hooking up of hardware to instrumentation. Even so, the approaches using peripheral forwarding are more complex than FDP. This is due to the prior requiring knowledge of the framework used to facilitate the setup and of the workings of the firmware. This framework is often Avatar[2] or custom made. The VPM projects Jetset and µEmu do not require hardware. Nevertheless, they are made complex due to requiring the memory layout, entry address, and goal address along with the firmware as input. In contrast, FSR projects are very simple to use. They only require the user to provide appropriate firmware to emulate. All steps from extraction to emulation are then done automatically.

- Risk: The amount of risk is relative to the fidelity of the environment presented to adversaries. As such, PF presents almost no risk as it does not allow for real time sessions with humans. Similarly, µEmu is unsuitable for human interactions as it dynamically switches between training its peripheral model and emulation. Both FDP and FSR present interactive, high fidelity environments and thus present a high risk of abuse. However, while malware in FDP situations is constrained by the hardware's capabilities, FSR emulators have more resources that can be abused by running on stronger host hardware. Lastly, VPM is highly dependant on the actual peripheral being emulated. For example, it is possible that the emulated virtual peripheral is not often used or not critical for the operations performed by malware. As long as the peripheral ensures that the firmware does not end up in an error state, its actual fidelity does not matter. Instead, the emulator's capability of handling the rest of the firmware, and the malware do.

- Fidelity during usage: The environments presented to adversaries by PF and VPM have a low fidelity. PF is not suitable for interactive sessions with humans due to the delay introduced by it. This is the result of having to communicate between software and hardware through a framework and non-conventional hardware channels. On the other hand, VPM's goal is not the creation of peripheral models that return correct data, but data adhering to certain logical constraints. Additionally, the peripherals' fidelity drops after the point up to which they were trained. In contrast, FDP has the highest possible fidelity as it provides fully working firmware backed by actual authentic hardware. FSR attempts to provide an interactive and functional environment on a surface level. Yet, compromises must be made due to the lack of hardware. For example, the camera feed will understandably not exist when emulating an IP camera. This hampers the overall fidelity of the approach.

- Fidelity of logging: Each "angle" of a system describes events differently. To increase the fidelity of captured data, the amount of sensors must be increased. Adding said sensors is simple in a virtual environment. This results in high fidelity for all approaches virtualising at least a part of the firmware. That is, all but FDP. Emulators can be hooked into by the host system. The approaches using peripheral forwarding do this already as they are built upon the idea of capturing and processing events. A virtual environment allows for introspection unlike a closed off, physical system. It is for this reason that FDP's data capture fidelity is lower. The data going through the proxies is only superficial, e.g. web requests and responses. Capturing the lower level events in order to study the firmware is hard to impossible, depending on the feature. For example, it could be possible to capture signals with a logic analyser and reverse engineer the workings of some module. However, capturing the resource usage of specific processes would be harder with only hardware access.

- Scalability: The scalability of a honeypot can be defined as the ratio of possible amount of simultaneous sessions to amount of investment. It is thus inversely proportional to cost and complexity. FDP, PF, and the VPM projects PRETENDER and Conware incorporate hardware in their setups. To some extent, it is possible to scale up one piece of hardware to multiple sessions. For example, FDP can be set up such that one device has multiple proxies pointing to it. However, hardware will always limit scalability. It must be acquired first and foremost. Then, it must be hooked up to instrumentation. This is less complex for the FDP approach than it is for the peripheral forwarding ones. We thus rate FDP's scalability as medium instead of low. The fully virtual VPM projects also have low scalability as they require engineering effort to determine the input parameters. Note that the scalability of all VPM techniques becomes high during usage, as already modelled virtual peripherals can be used without the original hardware. During emulation, only the lack of the host's resources could limit the scalability. Yet, as mentioned during the discussion of cost, the emulated

hardware is low end and thus does not require a great amount of resources. This also applies to FSR. As a result, it has high scalability. Deploying duplicate emulations of a firmware is viable.

- Maintenance: Honeypots require maintenance just as any other service. This includes providing resources such as electrical power, updating software in order to lower risk of abuse, and resetting the honeypot periodically to ensure adversaries do not alter the environment as it could affect future interactions. As such, maintenance is yet again a characteristic influenced by the amount of hardware involved. It is simple to restart an emulated environment and reset it to its original state, based on an image. Automating this process is also feasible for the same reason as extensive logging is possible in virtual settings. On the other hand, approaches incorporating hardware during usage complicate maintenance. Hardware may end up in an invalid state or store malicious data. Resetting it is not always evident. For example, some devices require users to press a reset button while others require the removal of a power source. The honeypot operator must also manually determine when to reset a system. As such, this process is hard to automate and hampers scalability as well.

Criteria	Full device proxy	Peripheral forwarding	Virt. peripheral	Full re-hosting
Cost	↑	↑	↑ (model) / ↓ (use)	↓
Complexity	∼	↑	↑ (model) / ↓ (use)	↓
Risk	↑	↓	peripheral dependant	↑
Fidelity (usage)	↑	↓	↓	∼
Fidelity (logs)	↓	↑	↑	↑
Scalability	∼	↓	↓ (model) / ↑ (use)	↑
Maintenance	↑	↑	↓	↓

Table 3.2: Relative comparison of different firmware re-hosting techniques using the characteristics of a honeypot. A limited amount of hardware is assumed to be available to the honeypot's operator. The symbols ↑, ∼, ↓ mean high, medium, and low respectively.

In conclusion, the current state of the art implementations of PF and VPM do not qualify for building a honeypot. Their major drawback is their low fidelity. PF's lack of fidelity is due to the delays when interacting with hardware. As for VPM, the lack of fidelity is the result of how the peripheral models are created. While VPM is not suitable for honeypots, it might work well for dynamic analysis. The reason being that the researcher would know the nature of the system and interact with it by means of automated tooling. Despite low logging fidelity and limited scalability, FDP is still a viable option for building high interaction honeypots. It is possible to keep costs and maintenance down if only a few devices are studied. The benefit of using FDP is that it allows adversaries to interact with unaltered firmware running on the original hardware. On the other hand, FSR provides much higher scalability and incurs less cost. The flexibility of a fully virtual environment allows for extensive logging and tweaking of the presented environment. The downside of FSR is that it only supports Type 1 systems as opposed to FDP, which supports any hardware as long as requests and responses can be proxied to it. Additionally, obtaining firmware images for FSR is not trivial. And even then, emulation of the firmware might not succeed.

Chapter 4

Implementation

The goal of this thesis is to study the current threats to IoT devices exposed on the Internet. This requires collecting data on adversaries' actions. Both automated and manual attacks are of interest to us. The prior represents the main threats given the scalability of the approach. The latter is of interest due to the high effort required by adversaries to perform such attacks. Manual attacks should present themselves to be intricate and targeted. Unfortunately, high interaction IoT honeypots are not readily available. Therefore, we build one ourselves using a state of the art firmware re-hosting technique.

This chapter explains our honeypot implementation. First, a schematic overview of the complete setup is presented, along with a brief explanation of the workings of each component. Then, we discuss how the components and related implementation choices help in fulfilling the requirements of the decoy and captor. To finish, encountered complications and possible improvements are discussed.

4.1 System overview

A schematic overview of the complete system with all related resources and services is shown in fig. 4.1. We end up using full system re-hosting, as will be discussed in section 4.2.1. As a result, all services are virtual and the whole setup can be deployed in a VM. This enables portability and scalability. The actual services acting as honeypots run in Docker[1], a containerisation framework. This has two advantages: precise modelling of the "fake", honeypot private network, and segregation between the services on the host and inside the honeypot network. In other words, it allows us to hide the captor part of the honeypot, i.e. instrumentation, from the decoy. This lowers the possibility of adversaries discovering the nature of the system, and consequently harming the integrity of the captured data. A practical example of this segregation is the network tap. This service captures all network traffic in the `192.168.0.0/24` subnet, which is the Docker honeypot network. This service could have been placed in a Docker container and attached to the network in question. However, the tap would be exposed in the event that an adversary gains a foothold in the private network and proceeds to scan the subnet. Unfortunately, running an emulator in Docker is not quite performant. We attempt to remedy this by performing the preparatory, iterative emulation phase of the emulation framework on the host. This phase must only be performed once to determine the configuration that enables a working emulation of the given firmware. The learned configuration can then be copied into the Docker container and used by the emulator running inside it.

4.1.1 Honeypot services

The Docker network contains two types of honeypots: high fidelity consumer firmware emulations and low fidelity MQTT simulations. The high fidelity devices are the IP camera D-Link® DCS-700L with firmware version 1.03.09, and the router Netgear® R7000 version 1.0.11.116_10.2.100. Both IP cameras and routers are prevalent IoT devices found in home networks. However, the ultimate reason for using these firmwares in particular, is the complications encountered in finding firmware of which the emulation succeeds. This is discussed further in section 4.3.1. Nevertheless, both devices run interesting services by default that we expose to the Internet. The IP camera presents a Telnet server on port 23 and

[1] `https://www.docker.com/`

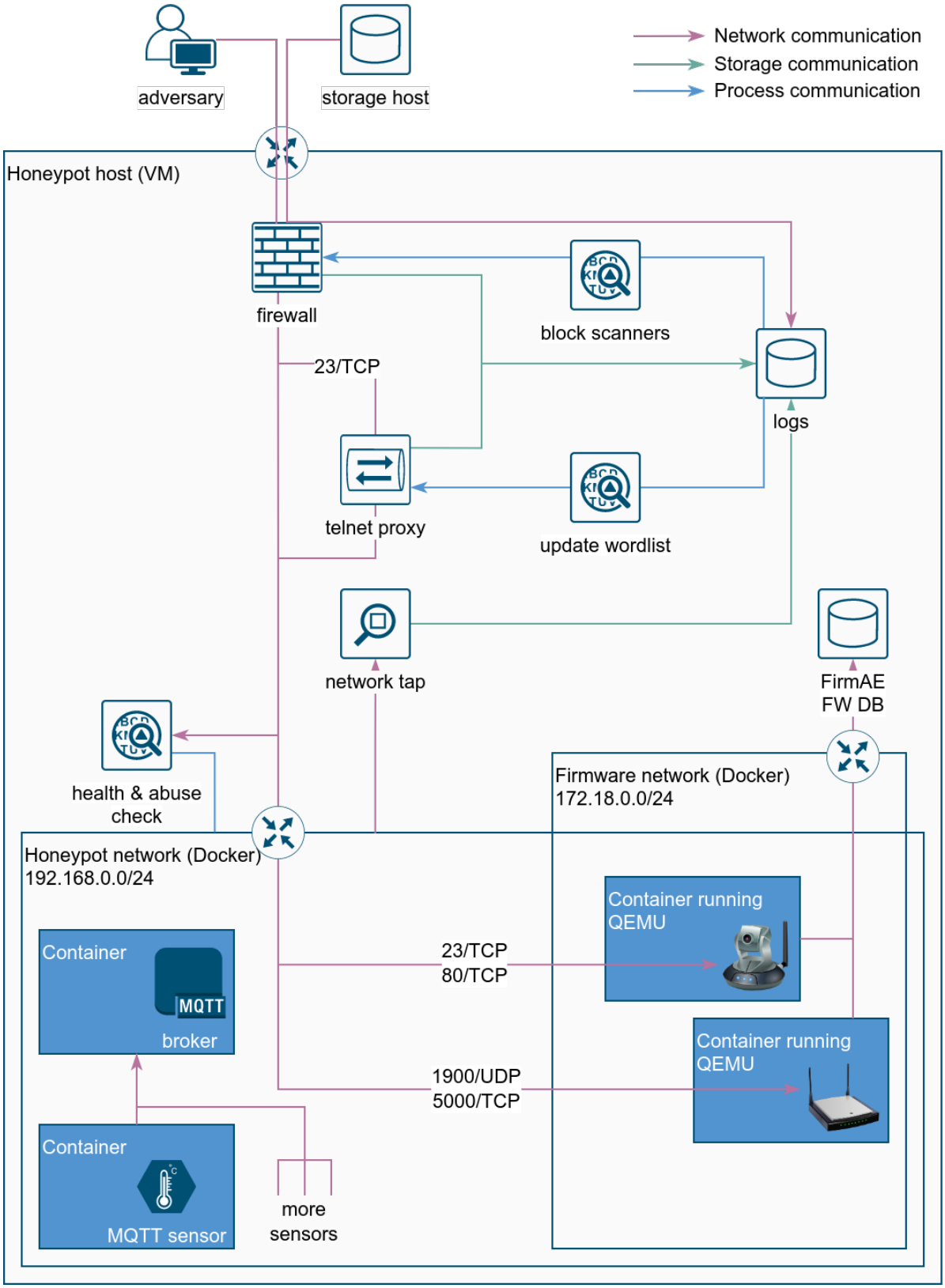


Figure 4.1: A schematic overview of the built system with all its components. Data flow is shown using coloured lines connecting the involved entities. Client-server relations are indicated by arrows.

an administrative web panel on port 80. A proxy is placed in front of the Telnet server. This has multiple uses. First, it intercepts scans coming from the outside, thus preventing the emulator from being flooded, which could result in possible downtime. Second, it enables data capturing by logging all Telnet interactions. Lastly, it allows authentication using multiple credential combinations. This allows us to implement honeytokens. As for the web server, it has a lower barrier to entry than a Telnet connection. Additionally, it is not known to contain any vulnerabilities in this version of the firmware. Meaning, it also has a lower risk of abuse than an interactive shell. Using this, we wish to discover what adversaries are interested in when interacting with an administration panel. Access to it is available using HTTP basic authentication with the common default credentials of `admin` for both username and password. The web server is supposed to show the camera feed and allow for configuration of the IP camera. However, making it believable required some work due to the limitations of the emulation framework, as discussed in section 4.2.3. The router also runs a web server, but we do not allow access to it from the Internet as that would be redundant. It also exposes ports 1900 and 5000 however. These are both parts of its UPnP service. Adversaries are free to interact with the service as they wish due to UPnP not implementing any authentication and authorization, as mentioned in section 2.5.2. Both UPnP and associated vulnerabilities are not new, with Squire presenting the `AddPortMapping` protocol abuse vulnerability in 2008. The intent of exposing it then is to gauge the interest in, and knowledge of, older technologies.

MQTT was chosen as communication protocol for the other services. The reason being that it has consistently been the most popular IoT application protocol for years, according to developer surveys [iot21] and search trends (see fig. 4.2). The MQTT devices are only accessible from inside the honeypot network. The goal of this part of the setup is twofold. First is determining adversaries' knowledge of exploiting the MQTT protocol. The second intent is to study the interest of adversaries in moving laterally through a network. In other words, will an attacker try to move through a private network after gaining initial access to an IoT device in said network? Only being accessible via the emulated devices is also the reason why the MQTT devices are low fidelity. An adversary whom gains access to the IP camera or router and then proceeds to scan the private network shows to have sufficient knowledge of how to interact with the unique Linux environment offered by IoT devices, e.g. using BusyBox and cross-compiling their custom tools to work on a RISC architecture. Making them prove themselves again for the internal part of the network is unnecessary. The MQTT clients are implemented using traffic generating scripts. These pretend to be either an air conditioner, smart thermometer, or smart lock. For the broker we use Mosquitto[2]. Several vulnerabilities are introduced based on the discussion of the protocol in section 2.5.1. No encryption or authentication is configured to facilitate adversary interactions. Furthermore, the broker allows clients to subscribe to all topics using wildcards. This enables traffic sniffing. Also, the topics to which clients publish follow similar structures. Uniqueness is introduced by having one level of the topics be a UUID. This UUID differs between each device. It is also their `ClientID`. Thus, an adversary sniffing traffic is able to discover all clients' IDs. By starting a new connection with the broker and providing a sniffed ID, it can be used to DoS or impersonate the original client. Last but not least, some of the clients periodically send the broker a request to check for updates. The broker is set to respond with the message `{"run": "date '+%Y-%m-%d_%H:%M:%S' > /etc/last_update_check"}`. That is, a shell command to be ran. While in practice nothing is actually being executed, we attempt to trick adversaries into thinking a command injection vulnerability exists.

4.1.2 Instrumentation

Instrumentation includes all supporting services. First we mention the FirmAE database and associated Docker network. FirmAE is the firmware re-hosting framework that is used. It is discussed further in section 4.2.1. The database is used to store properties of emulated firmwares and is thus essential to the framework's functioning. The emulation running in a Docker container must have access to the database. At the same time, the database must not be exposed to the honeypot network. We created a separate Docker network to solve this issue. Only containers running emulations and the database are part of it. By only routing the honeypot network interface through to the emulator, and not the database network one, it is ensured that adversaries can not access the database. Adversaries are unable to access the container as this would require escaping the emulator, which is not impossible but nonetheless highly unlikely.

[2] `https://mosquitto.org/`

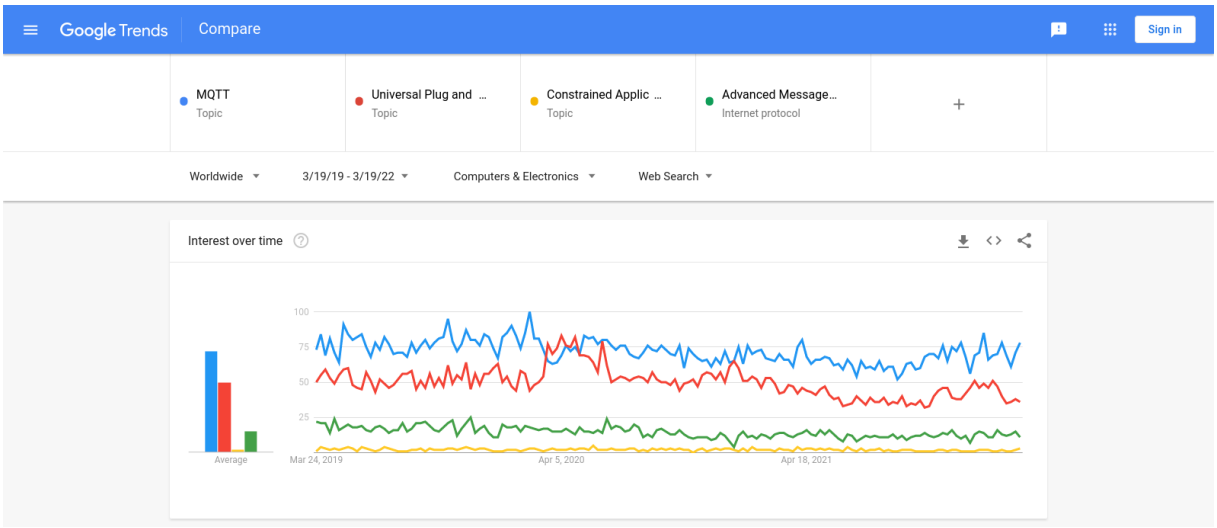


Figure 4.2: Amount of searches on Google for the application protocols MQTT, UPnP, CoAP, and AMQP. The graph goes back three years from the time of writing. The relative ranking has stayed almost completely static, with a small burst of interest in UPnP around mid 2020.

The health and abuse checking service is created to ensure maximum uptime and lasting fidelity for all honeypot services. Abuse of the honeypot is highly probable. One concern is that adversaries launch processes that take up all resources. Another possibility is them attempting to take over the device by e.g. stopping or re-configuring certain services. Both of these can result in downtime of the honeypot. The health and abuse checking service combats this. It periodically scans all honeypot containers and compares the running services to the intended list of services. It also checks whether resource are being overused. If so, it reboots the honeypot. The emulations require a clean reboot from their firmware image to ensure the clean-up of artifacts potentially left behind by adversaries.

The firewall is implemented using `iptables` and is essential for data control. It opens only the required ports and routes all related traffic to the emulated honeypot devices. Additionally, it enables bandwidth throttling using the `hashlimit` feature. On top of all this, it also enables the creation of a block list for IP addresses. The scanner blocking service uses this to dynamically block IP addresses of known legitimate Internet scanning services. It does this by reading the firewall logs and performing a reverse DNS lookup of the IP addresses. IPs belonging to a pre-set list of domains are blocked. Filtering out this legitimate traffic will make analysis easier. Of course, the honeypot operator can also manually add IPs. For example to block an adversary attempting to poison the honeypot.

The Telnet proxy has already been mentioned. As noted before, the proxy has support for multiple credential pairs. The system includes a service that dynamically updates this list of valid credentials. The reason for this is to create a believable environment. For example, allowing the username `admin` with a wildcard as password would be suspicious. Humans would notice how they are able to login with every password on their first attempt. A more believable approach would be to use a list of commonly used passwords. However, finding a comprehensive wordlist containing IoT-specific passwords is difficult. As to not exclude possibly interesting attacks, we create our own IoT wordlist. Initially, no password is valid. Over time, automated attacks will attempt to login using passwords known to them. These are usually default passwords for some IoT device. The wordlist updating service then periodically scans the proxy's logs and adds the captured passwords to the allow list.

4.2 Requirements and solutions

Above section gave a general overview of each component of the system and how they relate. This section will focus on the requirements to be met in order to appropriately implement a honeypot that can achieve the thesis' goals.

4.2.1 Consumer firmware emulation

The intent is to present a high fidelity environment based on consumer firmware to adversaries. This means firmware re-hosting is required. Looking back at our evaluation of techniques in section 3.4.5, two candidate approaches present themselves: full device proxy and full system re-hosting. Buying devices for a full device proxy approach would result in a lock-in situation given the associated costs. Instead, we opt for the full system re-hosting approach as it offers much more flexibility. This flexibility allows us to add and replace devices on a whim, which is a major pro given the experimental nature of research. However, this flexibility is drastically reduced by the efforts required to make the resulting environment believable, as discussed further in section 4.2.3.

Several projects following the full system re-hosting approach exist. As mentioned in section 3.4.4, these projects build upon each other's work. More specifically, Costin, Zarras and Francillon's and Firmadyne were created roughly around the same time, while FirmAE and Honware separately built upon Firmadyne's work. While Kim et al. does mention the possibility of using FirmAE to build a honeypot, the main goal of their paper is to improve the success rate of emulation, compared to Firmadyne, to enable the dynamic analysis of consumer firmware. They released their code as open source on GitHub[3]. Honware's focus, on the other hand, lies in increasing network reachability of emulated firmware in order to use the project to build a high interaction honeypot. This resulted in them setting up an online service providing honeypots on demand[4]. Unfortunately, they did not release their code publicly. Attempts to contact the authors of Honware were made, but no replies were received. As such, we settle on using the FirmAE project.

4.2.2 Visibility

Luring humans into the honeypot is important. It allows us to study more intricate attacks, and possibly observe interactions with the devices located in the back of the network. As such, the honeypot's IP address along with some credential pair must come to the attention of a human. Adversaries can obtain this data either directly, or indirectly e.g. via automated data gathering. We take several different approaches to advertising our honeypot based on the discussion had in section 3.1.3. No darknet alternatives were used due to ethical considerations. Each individual post contains the IP address of a honeypot instance, an indication that the ports 23 and 80 are opened, and a honeytoken made up of the username `admin` and a uniquely generated password. This unique password is added to the Telnet proxy's wordlist on the relevant system.

Our main approach is posting text snippets online. Initially, we posted manually to the web sites `https://pastebin.com`, `https://hastebin.com`, `https://controlc.com`, `http://pastie.org`, and `http://codepad.org`. GitHub gists used to be a popular alternative as well but, at the time of writing, it is no longer possible to post these anonymously. Posting once is also not enough. To increase the odds of being seen, multiple posts must be made. Thus, we wrote a script that posts four times per day. Unfortunately, it only posts to the first three mentioned web sites due to CAPTCHA restrictions on the other two. Adversaries gather information from these web sites using scrapers. These look for certain strings in the posted messages. We play into this by posting using several different templates. Two templates simply contain URIs to both the Telnet service and IP camera's web interface. One of these follows the standard format for URIs, i.e. `<protocol>://<username>:<password>@<ip>:<port>/<path>`, while the other places the credentials separately after the URI. The third template attempts to be easily machine readable by following the JSON format. This template is shown in fig. 4.3. Lastly, each template is accompanied with a slightly different title.

The second outlet for our advertisements is hacker forums. However, the story of selling leaked credentials, used by prior work, can not be adapted. The reason being that we only deploy two honeypot instances and consequently only have two IP addresses. Sharing such small amount as a sample would not be convincing. Several dozen additional unusable IP-credential pairs could be generated. However, the difference between the amount of working and invalid identities could also raise suspicion. As such, our cover story mentions finding an interesting system with exposed ports, followed by asking forum users to teach us how to hack it. By acting naive and providing only minimal information, we attempt to imitate an inexperienced individual. Similar requests for help are not unusual on the target forums. We thus argue that this simple story should be believable. The story is posted to `hackforums.net`,

[3] `https://github.com/pr0v3rbs/FirmAE/`
[4] `https://honware.org/`

```
// exposure_report.json
{
    "info": "exposure␣report",
    "time": <unix_time>,
    "device": [
    {
            "ip": "<host_ip>",
            "port": 23,
            "username": "admin",
            "password": "<generated_password>",
            "service": "telnet"
        },
    {
            "ip": "<host_ip>",
            "port": 80,
            "username": "admin",
            "password": "admin",
            "service": "httpd",
            "hostname": "DCS-700L",
            "device-type": "ipcam",
            "path": "/home.htm"
        }
    ]
}
```

Figure 4.3: The JSON message template used to advertise a honeypot instance. It pretends to be the output of a vulnerability scanner. The angle brackets indicate variables to be filled in appropriately.

`https://raidforums.com`, and `https://nulled.to`. However, `hackforums.net` automatically appends the results of a WHOIS lookup to messages containing IP addresses. This lookup shows that the IP is not a private address but that of a hosting provider, which is in conflict with us presenting the system as a home network.

Finally, we advertise the honeypot by making Google Docs® documents public. Similar to the hacker forums approach, the story surrounding the "leak" must be adjusted to match the context of our honeypot, i.e. it looking like a small home network. We pretend to share a document with our parents containing credentials for various devices at home. An example document is shown in fig. 4.4. It is not uncommon to store credentials in documents, despite it being a malpractice. The documents are publicly available from the Internet. We also share links to the documents on `https://pastebin.com` to increase visibility.

4.2.3 Believability

A honeypot's decoy must be believable to keep adversaries hooked. This not only means that the environment must look real but it must behave real as well. In other words, believability and preventing being fingerprinted go hand in hand. We will now discuss steps taken to trick users into thinking our honeypot is a real system.

The compromises made by full system re-hosting to emulate firmware hamper its fidelity. The FirmAE framework in particular was created with dynamic analysis in mind. The user interacting with the guest system is expected to be a researcher whom knows the true nature of it. As such, the framework makes no attempts at hiding itself. Numerous artifacts can be found in the resulting emulation, for example framework processes, files, and environment variables. An adversary could look these unusual artifact up and discover that they are not interacting with a real IoT device. A considerable amount of thought and effort must be invested into hiding the artifacts from all possible angles. However, actually doing so is relatively easy as the framework provides a modifiable shell script that hooks into the `init` process.

In Unix, process information can be found in the `/proc` directory. This is a virtual filesystem using procfs. Meaning, the files under this directory are not actual files stored on disk, but rather information

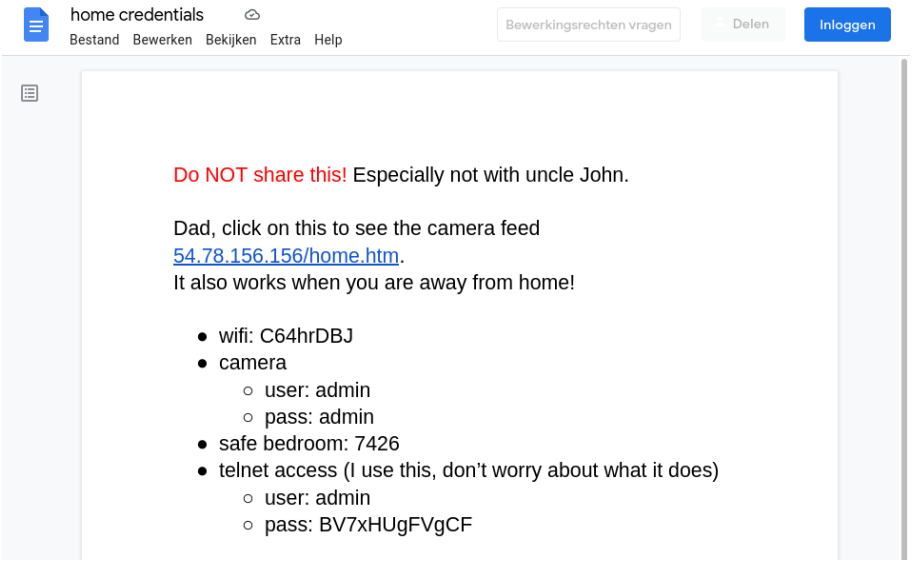


Figure 4.4: Google Docs® document containing unusable data such as a home Wi-Fi password, and information on one of our honeypot instances such as the IP address and a unique honeytoken.

gathered by the OS and presented with a file-like structure. Changing the visibility of processes requires manipulation of the data in said directory. We consider two approaches to hide sensitive processes from the eyes of adversaries. The first is remounting `/proc` using the `hidepid` option. This can be used to either deny access to process information, or completely hide processes from other users [Bow+09]. Due to being an option of procfs itself, this approach applies to `/proc` and thus the whole OS directly. As usual, the `root` user is unaffected as they have permission to do anything on the system. In other words, this approach would require us to create a least-privileged user to be used by adversaries. While commonly this is good practice, gaining root access might be an important part of exploitation. The alternative is to preload a library like the libprocesshider[5] project does. This library intervenes whenever data from `/proc` is read by tooling. It then filters the list of processes before it is passed to the invoking application. As a result, the filtered processes will not show in applications such as `ps`, `lsof`, and `top`, even when executed by the `root` user. However, the `/proc` directory is not directly affected by this. Listing its contents would thus still reveal all processes. We end up using the first approach. The reason being that it affects procfs directly, ensuring that we do not overlook anything that might still leak process information.

All framework files, such as the custom NVRAM library and `init` wrapper, are contained in a directory located at the root of the emulated guest's filesystem. These files are of importance to the emulation. Yet, adversaries seeing them might become suspicious. After some research, we found that the only practical way to hide files from the filesystem is to enter a chroot environment with all but said directory mounted in it. This also enables us to spoof system information such as the Linux version, or the fact that peripherals are virtual and provided by an emulator. This can be done by mounting a file with the spoofed data over the original. Lastly, the framework binaries also produce kernel logs. We configure `dmesg` such that these logs are hidden from users other than `root`. In conclusion, when an adversary authenticates over Telnet, they are dropped into a chroot environment as a least-privileged user. The login binary facilitates the hiding of environment variables. Meanwhile, framework binaries are executed by the `root` user, hidden from the least-privileged user by all means.

By definition, full system re-hosting has no access to hardware. In the case of the IP camera, this is clearly noticeable when visiting the administration panel as the camera feed does not load. The DCS-700L's camera feed works by pointing to a path containing an image, and reloading it every few seconds with client-side scripting. In the back-end, every few seconds the web server is supposed to call a device driver that takes a picture and replaces the image at aforementioned path with it. We could fix this functionality by writing a custom device driver. However, doing so would require huge reverse engineering efforts to make the custom driver compatible with the firmware. Instead, we decide to take several pictures from an IP camera at home and loop through them using a script. All private information is stripped from

[5] `https://github.com/gianlucaborello/libprocesshider/`

these images. Another symptom of lacking hardware is that data loaded from NVRAM is either empty or shows non-sensical values. Sane defaults for each NVRAM key must thus be set manually. Doing so requires us to reverse engineer the web server, matching each shown value to its respective key.

The lack of actual hardware also includes networking peripherals. FirmAE makes sure emulated devices are network reachable by setting up virtual interfaces for the emulator. The used settings are based on the configuration requested by the firmware during the emulation framework's incremental learning phase. Nevertheless, circumventing the original setup routine results in unusual configurations. We encountered misconfigured subnets for the guest devices' interfaces, along with missing default routes and DNS servers. This resulted in incorrect routing and no access to the Internet. Furthermore, interfaces used default MAC addresses. We applied fixes for all of the above. Devices may also assume certain static IP addresses and even hardcode these in their firmware. For example, a router in a private home network is likely to use the address `192.168.0.1`. But for our setup, this IP is reserved for the virtual router created by Docker. The honeypot Docker network is incidentally configured to use the `192.168.0.0/24` subnet, which is common for private home networks. We employ binary rewriting to ensure the emulated devices' IP addresses match that of their Docker containers throughout the whole firmware. Doing so has several practical constraints. First, IP addresses are commonly stored as strings in the firmware blob. This makes them easy to find. However, the replacement string may not be longer than the original to prevent overwriting of the trailing null-byte, or other bytes that follow. A shorter string may be used if it is padded to the original length with null-bytes. An incorrect edit may break the file structure and corrupt the whole firmware or one of its components. Second, firmware blobs may be quite big. The binary rewriting software must be able to process up to hundreds of megabytes of firmware efficiently. As a result, we implement the rewriter using a simple Flex[6] program.

4.2.4 Data control

Proper data control is required to decrease the risk associated with hosting high interaction honeypots. We already mentioned the workings of the health and abuse checking service in section 4.1.2. It combats abuse by periodically scanning and restarting the system if required. The Telnet proxy's wordlist could also be considered a data control scheme as it limits the amount of access to the Telnet service. The disk size and main memory assigned to emulated devices are both limited to 256MB. Also, the CPU usage of emulators is limited to a fraction of the host's possible performance. The amount is determined experimentally such that the emulator does not time out during boot or normal usage.

Nevertheless, arguably the most important data control component is the firewall. Handling ingress traffic is relatively simple. The firewall can block unwanted IPs such as those of legitimate scanners. The block list is created using the `ipset` utility and handled by a single `iptables` rule. This is done to minimise the complexity of handling firewall rules. All ports but those pointing to honeypot services are also blocked. As adversaries have no access to the firewall running on the host, they can not open new ports. This helps in preventing the spread of malware that the honeypot gets infected by. The reason being that, given this setup, our honeypot can not be turned into a server that the malware binary can be downloaded from. However, the ports of the honeypot services can be reused by halting said services. The infected honeypot can also act as the client and upload a sample to the next victim. As for further configuration of the firewall, bandwidth limitations on incoming traffic are unnecessary as we do not expect to become the victim of a DoS attack, given that there is nothing to be gained for the attacker. Yet, a limit is added to match that of egress traffic. Invalid packets are not dropped as they might be part of an adversary's attempt to exploit the honeypot.

With that said, handling egress traffic is more complicated as discussed in section 3.2.1. Blocked ports or protocols can easily be circumvented. Adversaries may also become suspicious of the legitimacy of the system based on selective blocking. As such, we opt to implement a heavy bandwidth limitation on all outgoing traffic. More specifically, each IP-port tuple is limited to receive a maximum of 256B/sec. This number is chosen such that interactive shell sessions feel slow but are usable for humans. Yet, as packets in e.g. a reflection attack can be as small as 400B [Maj17b], the possible contribution to attacks is practically nullified. The bandwidth limitation for the IP camera's web server is less strict as it contains bigger assets, such as the camera feed. Lastly, we mention how, during early testing deployment, our honeypot received an abuse report within several hours of being online. The reason being a small firewall misconfiguration. While this was quickly resolved, it highlights the importance of data control.

[6] `https://github.com/westes/flex/`

4.2.5 Data capture & collection

Logging is performed from multiple angles. We capture all traffic sent through the honeypot Docker network using `tcpdump`. This provides a full view of everything happening on a network level. As for the Telnet proxy, it logs all shell input and associated responses. More importantly however, it takes note of all authentication attempts, both failed and successful. The information it records includes the used credentials, source IP, time of event, and duration of a session. While the network logs already contain all the above, the proxy's format facilitates analysis. Last but not least, both MQTT devices and firewall log all events related to themselves.

While we do not expose any ourselves, we briefly discuss logging of services utilising encryption. These services complicate logging depending on the service in question. For example, a web server's TLS traffic can be decrypted with the appropriate certificates. These certificates are available to the honeypot operator as they possess a copy of the firmware. SSH traffic, on the other hand, employs session keys to provide forward secrecy. As their name implies, these keys are unique to each session and thus harder to acquire. Logging SSH traffic could be done via a proxy such as cowrie, or by logging events through a custom kernel. If required, both approaches would have been feasible given that our setup already uses a proxy for Telnet and a custom kernel for FirmAE.

Logs are rotated periodically. All services are configured to use the same date and time notation to ensure consistency between them. Consistency between honeypot instances is ensured by using UTC for each timestamp. Each log file also includes the name of the instance it was captured on. A host separate from the honeypot instances is responsible for long-term storage. It periodically pulls log files, resulting from rotation, to ensure secure storage.

4.3 Complications

Every endeavour comes paired with some difficulties. Previous sections already discussed minor issues that were encountered and how they were handled. This section will cover the bigger complications. These are either resolved with certain compromises being made, resolved and discussed for documentation purposes, or unresolved altogether. We discuss these issues to highlight the downsides of our approach and the used technologies. This information can be used as a starting point for improvements in future works.

4.3.1 Acquiring firmware

Full system re-hosting has the advantage of not requiring any hardware in principle. A firmware blob is all that is needed to get started. The Firmadyne project contains a scraper[7] for several popular vendors' web sites that downloads provided blobs. Thus, despite being limited to Type 1 devices, we did not consider obtaining firmware for various devices to be a possible complication at first. Yet, upon closer inspection, the offer turned out to be severely limited due to a multitude of reasons. First, some smaller vendors simply do not provide any downloadable files. Second, the firmwares provided are notably only for IP cameras, routers, access points, and switches. Third, we notice that the majority of firmwares are for older devices. We argue that the second and third reason go hand in hand. The rational being that they are the result of vendors starting to provide update functionality via companion apps. This removes the need for users having to manually download blobs from the vendor and upload them to their devices. Thus, to use both newer firmware, and firmware of more diverse devices, e.g. smart lights or thermostats, the vendors' apps would have to be reverse engineered. This would be a major undertaking. We thus consider it to be out of scope for this thesis. However, it can be a viable option for future work. Lastly, we also mention that vendors only tend to provide the latest version of a firmware. This is good practice as it prevents users from accidentally downloading an outdated, vulnerable version. Nevertheless, it does limit security research. For our project in particular, we were unable to procure any firmware with known CVEs that FirmAE is successfully at emulating, in order to gauge whether they are actively being exploited.

As just mentioned, once having acquired a firmware blob, emulation is not promised to succeed. We encountered problems in extracting the essential files from blobs. The main reasons being blobs containing only files needed to be changed by the update, and proprietary file formats. Emulation would also fail during the preparatory emulation phase as the framework could not arbitrate appropriately. Last but

[7] `https://github.com/firmadyne/scraper/tree/f20ca9a6cc/`

not least, some firmware samples would seem to boot properly and be accessible over the network. Yet, when actually attempting to interact with the emulated environment, we would notice that some critical services, such as the web server, would not be running. These are unresolved issues with the state of the art implementation of the full system re-hosting approach itself. As such, one must wonder whether using a full device proxy would not have been a better choice. This approach would have allowed us to provide a wider variety of devices that are known to work, though as discussed at a cost of scalability of the implementation.

4.3.2 NVRAM configuration

Ensuring that the emulated firmware has access to a working NVRAM implementation is an important part of full system re-hosting implementations. Yet, setting sane default values for each key requires manual work. The emulated IP camera fortunately contains only a handful of important keys. We are thus able to make its web server look believable with minimal effort. However, the required manual work might quickly become infeasible for firmwares that contain hundreds to thousands of keys. This is the case for the router that is emulated in our setup. It offers an expansive web server in several different languages. This results in numerous options and strings being stored in NVRAM. Note that this is only one of the services using the NVRAM on said device. We attempted to set default values for only the English language. This required reverse engineering of the web server in order to determine the used NVRAM keys. We discovered that presented NVRAM values on web pages match with indices in the pages' corresponding templates. When parsing a template, the web server uses the indices to access an array of strings containing the NVRAM keys. However, this array contains several thousands of keys. At this point, we abandoned this work as it would have required an unreasonable amount of engineering effort to set sane NVRAM values to make the emulation believable. We instead decided that the router's web server would not be needed as the IP camera already provides a similar service for adversaries to interact with. The router's UPnP service is exposed however. Setting sane values for it was easier as only a handful of keys are used. In conclusion, while full system re-hosting allows firmware to run without NVRAM hardware, it hampers fidelity due to values being missing out of the box. A possible solution would require dumping the contents of NVRAM from an actual device. This is the approach used by EMUX[8], a Type 1 firmware emulation framework that requires extensive manual arbitration and data acquired from an actual device.

As mentioned in section 3.4.4, the function declarations used to access NVRAM might differ from vendor to vendor. We encountered this issue in the IP camera's firmware. Both loading default values, and setting new values via the administration panel would fail. By tracing the web server as well as the `nvram_set` binary, available in the firmware, we discovered that the original function declaration for setting a value is `int nvram_set(const void* _, const char* key, const char* value)`. This differs from the default declaration provided by FirmAE in that it takes a third parameter in addition to the key and value. We thus applied this change to the custom NVRAM library and cross-compiled it to the IP camera's architecture. We describe this process to highlight that this is a weakness of full system re-hosting and that the approach sometimes requires great manual effort.

4.3.3 Technicalities of routing through Docker to QEMU

Running the IoT device emulations in Docker containers has several advantages, as mentioned before in section 4.1. It is also a necessity when attempting to run multiple emulations side by side, as the created virtual interfaces might otherwise interfere with each other. Yet, the incorporation of Docker in the honeypot setup also causes several technical challenges, the biggest of which being how to route traffic between the Docker container and the emulator hosted within. Or, more specifically, how to route network traffic from the Docker container's honeypot network interface to the emulator's virtual network interface, and vice versa. While routing in and of itself can easily be accomplished, we additionally wish to match the IP address of the Docker container in the honeypot network to that of the emulated device. This is in order to increase believability. An adversary scanning or moving through the internal network, i.e. the honeypot Docker network containing other honeypot services, would otherwise be able to notice the discrepancy between the IP seen by other devices and the IP shown inside the emulation.

Initially, traffic between the interfaces was being routed using `iptables` rules. This worked as intended while the interfaces had unique IP addresses. We then attempted to implement the matching address

[8] `https://github.com/therealsaumil/emux/`

constraint. Yet, as one might expect, having multiple interfaces with the same IP address results in networking conflicts. An alternative approach was considered that would involve removing the IP address of the emulator's virtual interface, and using certain rules to handle the routing. Unfortunately, `iptables` routes traffic by rewriting IP addresses. It was thus clear that this tool would not be usable for the given problem. However, the idea of removing the IP address of the interface led us down the path of traffic mirroring. The `tc`[9] Unix utility is part of the `iproute2` suite and is used for controlling traffic flow. It works on the kernel level and can thus send traffic directly to interfaces, without the need of network layer addressing. We use `tc` to redirect traffic between the two interfaces in both ingress and egress directions. This is accomplished using the command shown in fig. 4.5. The first command creates a queue and attaches it to the Docker interface, while the second defines how to manage the packets stored in said queue. This handling of packets comes in the form of a filter. It matches packets given certain rules and performs appropriate actions. In our case, all traffic is selected and redirected to the emulator's interface.

```
tc qdisc add dev <docker_interface> ingress
tc filter add dev <docker_interface> parent ffff: protocol all u32 \
    match u8 0 0 action mirred egress redirect dev <emulator_interface>

tc qdisc add dev <emulator_interface> ingress
tc filter add dev <emulator_interface> parent ffff: protocol all u32 \
    match u8 0 0 action mirred egress redirect dev <docker_interface>
```

Figure 4.5: Shell commands instructing the redirection of traffic between the docker and emulator interfaces. The commands are presented as two pairs, one for ingress and one for egress traffic. The variables in angle brackets are to be replaced by the interface names in question.

4.3.4 Hosting

Our setup is fully virtual and can thus be deployed on any host. We deploy two identical instances: one on an OVH Virtual Private Server (VPS)[10] and the other on an AWS EC2[11] instance. This allows us to capture more and diverse traffic. It also allows comparison between providers. For example, one provider might be targeted more often than the other. However, hosting on public providers might impact the believability of our honeypot. The reason being that both OVH's and AWS' IP ranges are publicly known. An adversary can easily perform a WHOIS lookup on our instances' IPs. This would expose them being hosted in a cloud environment, which is in conflict with the context of our setup that is a private network exposing IoT hardware. Supporting this concern, Dang et al. reports receiving 6.7% less traffic on their low fidelity IoT honeypots hosted on AWS, compared to other public clouds. Due to practical constraints this remains an unresolved issue for our implementation.

4.3.5 Effectiveness of advertisement

We extensively advertise our honeypot instances, as discussed in section 4.2.2. Unfortunately, measuring the effectiveness of the advertisements is complicated. The honeytokens' reach can be recorded in two locations: when being used, and at the location they are shared at. The prior entails the recording of interactions and filtering them based on the usage of honeytokens. This approach only reports on the set of tokens being used and is not an indicator of the population they have reached. Yet, recording of interactions is fully in the hands of the honeypot operator and can thus be done consistently. On the other hand, the latter approach consists of gathering analytics at the distribution venue itself. This measures the amount of people who interact with the resource directly. Gathering analytics on honeytokens at the venues they are shared at is unfortunately not always possible. The only venue we use with built-in analytics is `https://pastebin.com`. It reports the amount of unique views on a paste. Google Docs® had a similar feature but it has since been removed. In conclusion, we are able to measure the usage of honeytokens, but not whether our advertisements reach adversaries. This is an unresolved issue in our setup.

[9] `https://manpages.ubuntu.com/manpages/xenial/man8/tc.8.html`
[10] `https://www.ovhcloud.com/en/vps/`
[11] `https://aws.amazon.com/ec2/`

4.4 Conclusion

In conclusion, we built a honeypot exposing high interaction services by means of emulating consumer firmware. Network depth is added by means of low interaction MQTT simulations. The setup is fully virtual and consequently offers high portability and scalability. This allows it to be deployed on any host such as a VM or VPS. While high interaction honeypots are known to require a relatively high amount of effort to set up and maintain, configuring the firmware emulations to be believable proved especially labour intensive. Both this effort being required, as well as difficulties in acquiring firmware, raise doubts about the effectiveness of current state of the art full system re-hosting implementations for building honeypots. Complementing the approach with data from a hardware device might be a viable option.

Chapter 5

Data analysis

Up till this point we have studied IoT security theoretically, learned about firmware re-hosting, and built a honeypot. Knowing whether the resulting product is believable and effective at capturing malicious interactions, as well as studying the current state of IoT security in the wild, requires the analysis of the logged data. This chapter is dedicated to that end. The first section is used to orientate by creating a general overview of the captured traffic. The three following sections are intended to analyse the captured data related to each of the three main services exposed by the honeypot instances, namely Telnet, web, and UPnP. In each section, general trends are discussed first before zooming in on peculiar events or the workings of specific artifacts. Last but not least, a section is dedicated to discussing whether adversaries show interest in spreading through private networks.

As mentioned in section 4.3.4, two identical instances of the honeypot are deployed. The OVH instance was used for experimental purposes. On the 22nd March, 2022 it was converted to only allow logging in by means of honeytokens, disabling the dynamic wordlist generation and the usage thereof. The reasoning being that running malware might be interfering with human access via honeytokens. On the 5th April, 2022 the OVH instance's Telnet proxy was disabled. Any interaction with the instance's Telnet service was from now on directly with the IP camera's service. Logging in was only possible with `admin` as both the username and password. Unfortunately, the AWS instance experienced technical issues between the 4th and 10th February, as well as between the 5th and 21st March, 2022. The honeypots were deployed for production on the 27th January, 2022. The AWS instance's network tap service failed, resulting in no logging of the traffic within these time intervals. However, the Telnet proxy continued to log as intended. This proves the importance of a multilevelled approach to logging. The cut-off date for data to be included in the analysis is the 31st March, 2022. This results in a timespan of 64 days being analysed. However, the honeypots were kept online for a longer duration. This allowed us to analyse already collected data, while simultaneously being on the lookout for noteworthy events that deviate from the norm. Figure 5.1 shows the timeline of the experiment along with the relevant events.

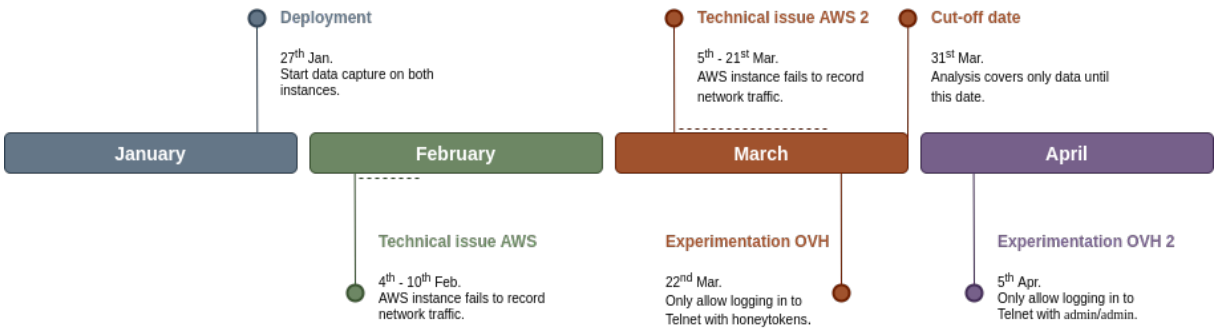


Figure 5.1: Timeline of the events that occurred during the experimental phase of this thesis.

5.1 Protocol interest

To start, statistics on the used protocols are extracted from the traffic captured with the network tap. This gives an overview of all interactions made with the honeypots over their lifetime. All incoming and outgoing frames are plotted. These are categorised based on their application layer protocol, as determined by Wireshark[1]. As for outgoing frames, note that these are the number of frames generated by the honeypot instances before they hit the firewall. The actual amount of frames leaving the instances is reduced severely by the bandwidth limitations set in place. A log scale is used to indicate the amount of frames, i.e. for the y-axis. This is due to exceptionally high peaks of traffic at certain times. Additionally, a high variety in protocols is identified by Wireshark. However, most of these register less than a hundred frames per day. As such, only the top five protocols per instance in terms of frames are shown. All remaining frames are grouped under the "other" label.

Looking at the incoming application layer data in fig. 5.2, we immediately notice that the ranking of protocols matches the findings of Metongnon and Sadre [MS18]. This confirms the validity of our captured data. More specifically, Telnet is by far the most popular protocol and is so consistently. Similarly, HTTP is also consistently popular. The difference in amount of captured frames between Telnet and HTTP, and HTTP and other protocols is substantial. For example, HTTP receives more traffic than other protocols by roughly a factor of ten. Note however how HTTP peaks from time to time, surpassing even Telnet on those days. Speaking of peaks, SSDP generally receives only around a hundred frames per day. Yet, on some days up to 10^6 frames were captured. Remarkable is that trends match between the instances. This applies not only to the amount of frames per protocol, and thus implicitly their ranking, but also to certain events. Days with peaks in a protocol roughly match between both instances. Lastly, note the four peaks of DNS traffic on the OVH instance. These anomalies are worthy of further investigation.

Figure 5.3 shows the outgoing traffic on the two instances. Here the traffic labelled "data" competes with Telnet in number of frames. It is labelled as such as the frame's payloads contain no apparent structure conforming to any known protocols. The honeypot implementation does not run any service that generates this kind of traffic as it would only complicate analysis. Thus, it must be data of an unknown kind created by the malware samples that infected the instances. The graphs also show several peaks. These include several for SSDP matching the incoming SSDP peaks, as expected. Furthermore, peaks in outgoing DNS traffic match the peaks of incoming HTTP traffic. This makes sense if the incoming HTTP data is seen as the responses to outgoing requests, which in turn require DNS lookups. Both the EC2 and OVH instance's graphs also show big, but infrequent, peaks of Google Quick (GQUIC) and OpenVPN traffic respectively. Contrary to the previously mentioned ones, these do not directly match any peaks in incoming frames.

We explain the notable occurrences. Telnet, HTTP, and UPnP/SSDP are discussed in section 5.2, section 5.3, and section 5.4 respectively as they match the exposed honeypot services and thus require more in-depth discussions. The reason for the unusual peaks of certain protocols, such as DNS and GQUIC, can be explained more briefly. Figure 5.4 shows a sample per peaking protocol. Wireshark indicates each of the shown frames to be malformed. Inspection of the parsed protocol fields confirms this. For example, fig. 5.4a shows supposed OpenVPN data with a network time in the year 2029. This is nonsensical as the recordings were made in 2022. Similarly, the GQUIC packet is not valid either. The shown packet's flags indicate that it is a reset packet. According to early drafts [Ham+16], which describe GQUIC more appropriately than the QUIC RFC which includes numerous improvements to the protocol, a reset packet should contain the tags PRST followed by RNON, RSEQ, and CADR with their respective context specific values. Tags are bytes in the payload used to distinguish variable values. However, as seen in fig. 5.4b, the packet's tags are not readable. The faulty classification is due to the frames being sent over UDP. UDP lacks the concept of connections. As a result, Wireshark can not use prior frames to infer application protocols and instead guesses based on other information, such as used port numbers. For example, the source port shown in fig. 5.4c is 53. Thus, the frame is labelled as DNS. Yet, that frame contains an SSDP request as payload. And indeed, the peaks of incoming DNS traffic to the OVH instance in fig. 5.2 coincide with peaks of SSDP traffic. This leads us to conclude that the adversary purposefully crafted the packets to be misidentified. The reason being to confuse and evade defensive tools. In the example of fig. 5.4c using port 53, this would be circumventing firewall rules as UDP for said port is rarely blocked to allow DNS traffic to pass through.

[1] `https://www.wireshark.org/`

Last but not least, a trend in the amount of Telnet frames can be observed upon closer inspection of the graphs. The amount of outgoing Telnet frames on the OVH instance slowly increases from 10^6 to 10^7. The increase starts roughly halfway through the time period. Looking back at fig. 5.2, this trend can be observed for incoming frames as well. Insufficient data is available to draw any definitive conclusions on the reason of the increased interest in Telnet within this short time period. We hypothesise that it could be (indirectly) related to the change in political climate as a result of the war in Ukraine, which started on the 24th February, 2022. This date roughly matches the start of the increase in interest. Further investigation in future work would be required to draw definitive conclusions.

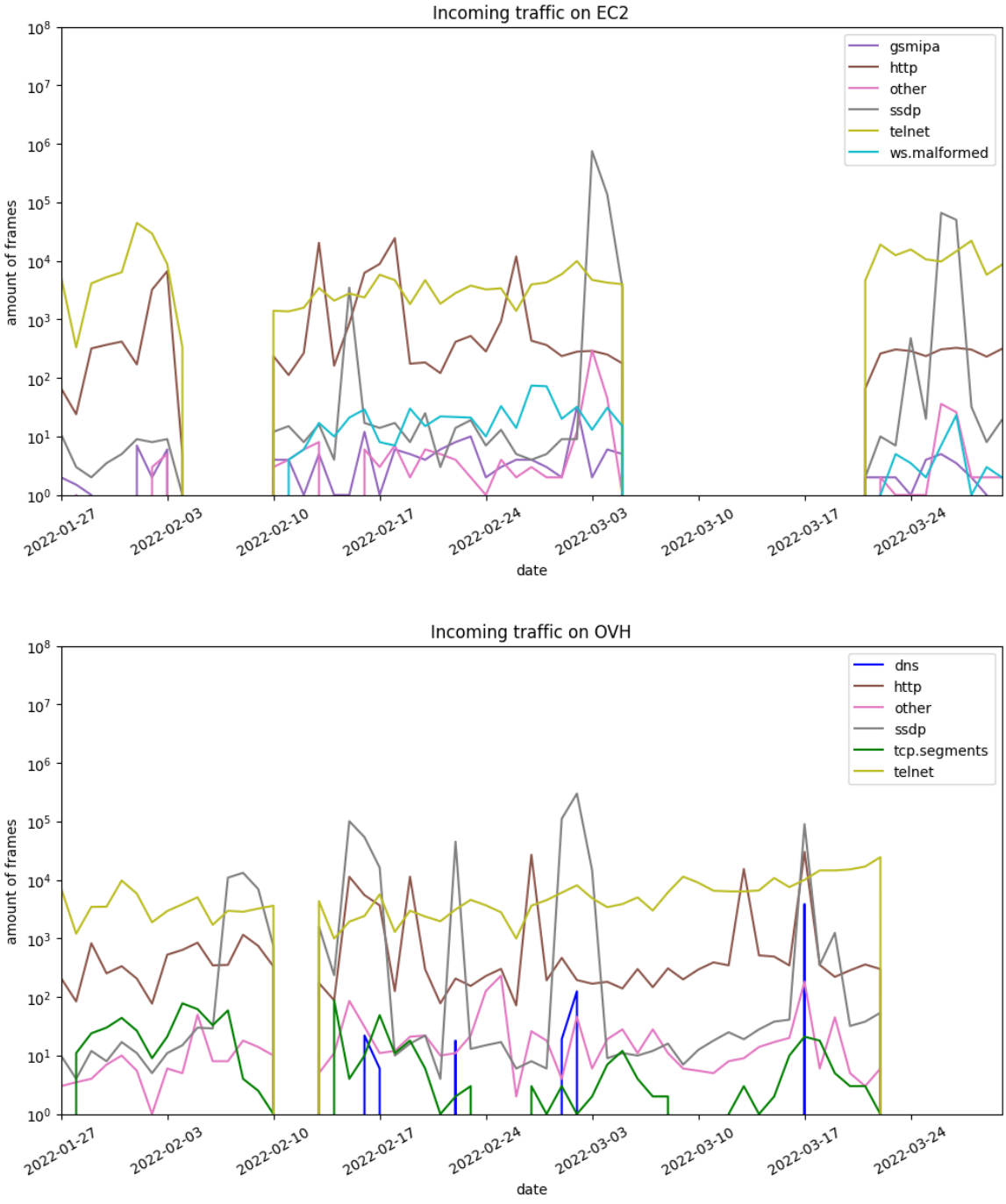


Figure 5.2: Incoming application layer traffic, per instance, over the analysed duration.

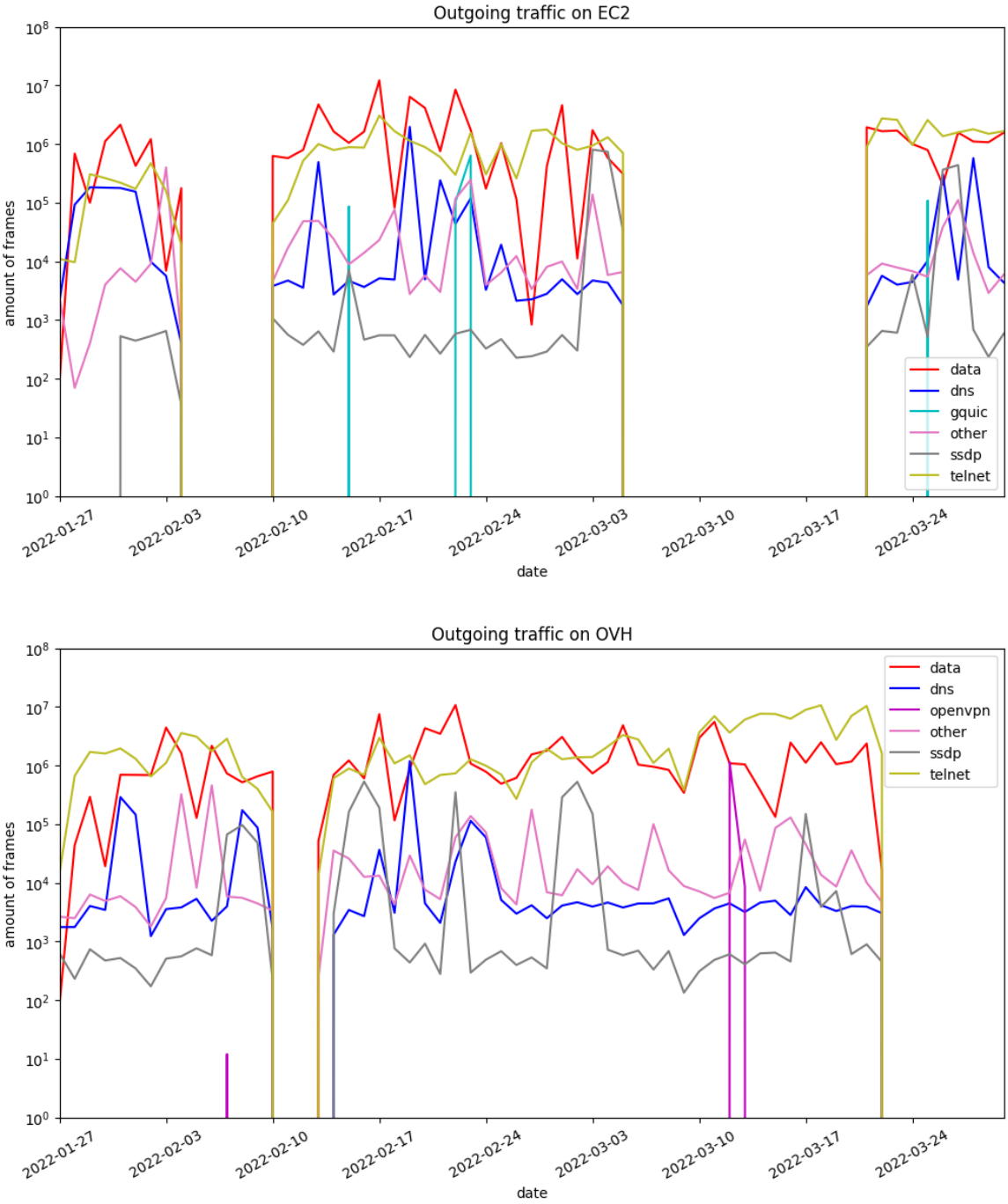
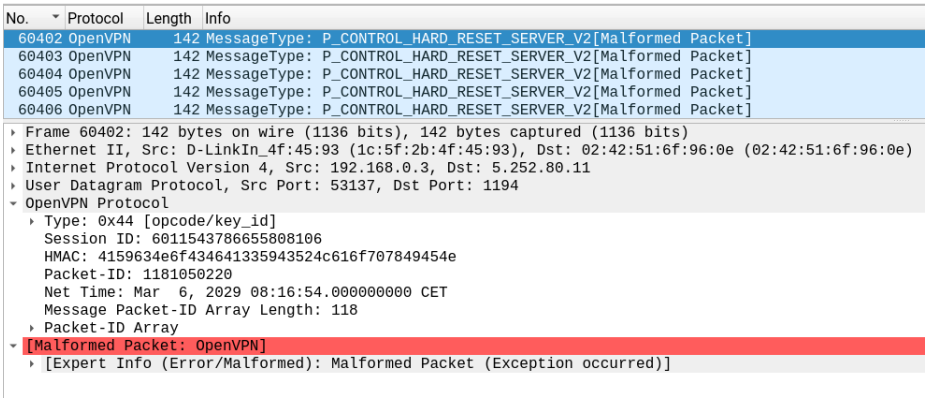
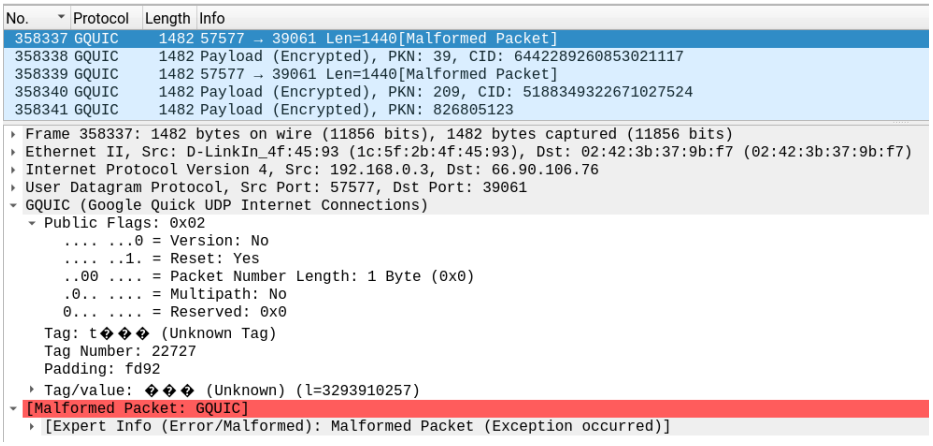


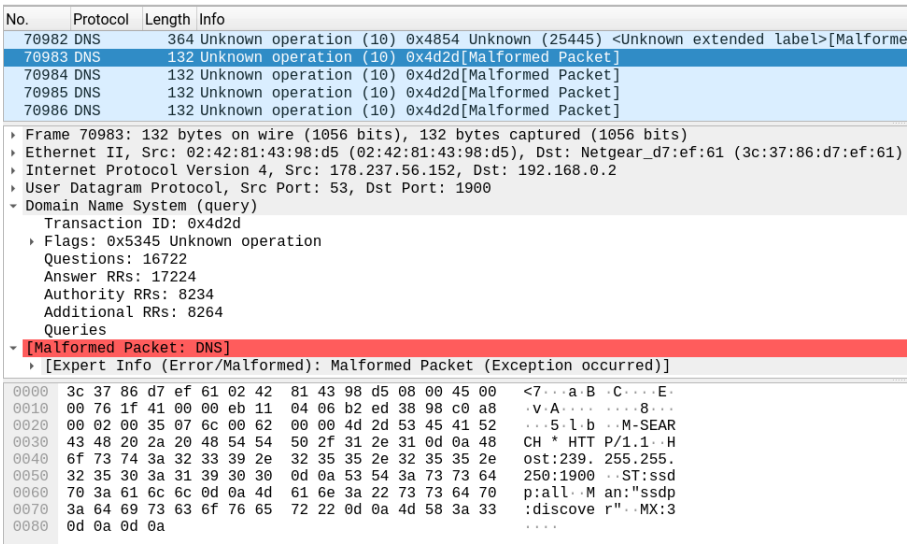
Figure 5.3: Outgoing application layer traffic, per instance, over the analysed duration.



(a) Frame wrongly marked as OpenVPN traffic. The parsed properties, e.g. IDs and the time, are nonsensical.



(b) Frame wrongly marked as GQUIC traffic. The parsed properties, e.g. the tag, are corrupted.



(c) Frame wrongly marked as DNS traffic. The payload contains plain ASCII data that can be recognised as an SSDP request.

Figure 5.4: Samples of frames associated with the unusual peaks in certain protocols observed in fig. 5.2 and fig. 5.3. These frames are categorised under the wrong protocols, as indicated by Wireshark in red, which causes the peaks.

5.2 Telnet access

Telnet is the most popular IoT protocol being scanned for on the Internet. Appropriately, it is the biggest graphs in the figures in section 5.1. Our honeypot Telnet service provides direct shell access after authentication. However, as the environment is constrained, not all commonly used Unix tools are available or only available via BusyBox. Additionally, custom binaries transferred onto the honeypot must be compatible with its architecture. These are some of the considerations expected to be handled by adversaries. This section is divided into three parts. The first part performs an analysis based on general statistics. The second part dives into the workings of shell commands resulting from certain interactions. The last part is dedicated to analysing captured malware samples.

5.2.1 Statistics

We analyse the Telnet proxy's data and extract statistics to create table 5.1. Note that rows with units pertaining to uniqueness do not simply sum up into the "Total" column. Rather, the uniqueness is calculated over the data of both instances in said row. Given the resulting numbers, the first observation is that attacks overlap between the two honeypot instances. We also remark that the ratio of failed to successful login attempts for the OVH instance is skewed compared to that of the EC2 instance due to the earlier mentioned experimentations.

Statistic	AWS EC2	OVH VPS	Total
Unique IPs	3632	3084	5957
Login attempts	87724	131879	219603
Failed logins	67714	116079	183793
Successful logins	20010	15800	35810
Unique passwords	475	558	609
Average IPs per password	110	125	201
Average unique IPs per password	48	51	81
Sessions without commands	3997	1824	5821
Sessions with commands	16013	13976	29989
Sessions with unique command sequences	734	863	1365
Median session duration (seconds)	11.7	13.4	12.6
Median words per session	105	107	106
Unique IPs mentioned in sessions	75	113	141
Captured binaries	4392	4320	8712
Captured unique binaries	236	306	420

Table 5.1: Statistics on interactions captured by the Telnet proxy, categorised per instance.

First, notice that the amount of unique IPs is smaller than the total amount of successful logins. This means that remote devices periodically re-visited the honeypots. Grouping logged shell commands by IP address reveals one of the reasons for this. Namely, sessions with the same remote IP address are likely to contain subsets of the same command sequences. More precisely, the sequences start identical but some of them close the connection abruptly. It is unlikely that the sessions are terminated intentionally as a result of the malware determining the instances to be honeypots. The reasoning being that the disconnects are too sporadic and that other sessions from the same IP addresses do eventually terminate properly, i.e. drop and run malware binaries. A more likely explanation is network failures, or timeouts due to slow processing given the inherently weak computational power of IoT devices combined with potentially other malware already running.

Second, we delve into the used credentials to access the Telnet service. Remember that our setup uses a dynamic wordlist building technique, as explained in section 4.1.2. The list starts empty. Passwords are then automatically added after they appear in the logs. Although several thousands of unique IPs visited the honeypots, only a few hundred unique passwords were registered. This is notable given that theoretically the amount of credential pairs is infinite. In other words, the wordlists used by adversaries are relatively small. Additionally, different malware implementations use (sub-)sets of the small amount of collected passwords, and/or a large amount of the unique IPs that visited us were infected by the

same malware. This can be confirmed by the large amount of IP addresses per password. Even after compensating for sessions initiated by re-visiting IPs, many of them use the same passwords to access the honeypots. The amount of IPs per password, both with and without duplicates, is also higher in total than it is per individual instance. Meaning, attackers that exclusively visited one of the instances share passwords with others. Furthermore, none of the distributed honeytokens have been used to access the Telnet services. The only used outlet capable of reporting statistics is `https://pastebin.com`. All honeytokens posted on this website are visited around 14 times. We theorise these visits to come from scrapers as all pastes have been visited roughly the same amount of times. Thus, despite using several stories and formats, and publishing on various platforms, we were unfortunately unable to incite any human interaction using the created advertisements. Lastly, the top 10 captured passwords are shown in table 5.2. One can identify these to be common default passwords. Although `vizxv` and `xc3511` seem to stick out at first, they are default passwords for Duhua© security cameras and HiSilicon© Digital Video Recorders (DVR) respectively.

Rank	Password	Unique IPs using it
1	`vizxv`	1488
2	`admin`	1485
3	`123`	1246
4	`12345`	1221
5	`root`	1033
6	`daemon`	953
7	`default`	944
8	`1234`	937
9	`xc3511`	864
10	`user`	830

Table 5.2: Top 10 passwords based on the amount of unique IPs using them to access the honeypot instances.

Further analysis of the numbers in table 5.1 shows that the median duration of a session is only several seconds. Meanwhile, the median amount of words per session is slightly over a hundred. From this it can be concluded that the majority of interactions are automated as a human is unlikely to be able to type at such speed. Automation of exploitation is highly important for adversaries and is certainly not a new concept. The reasoning being that every infected device adds more computational power to the adversary's network. Yet, in the case of IoT devices this is doubly important as each device provides only a small amount of resources due to low hardware capabilities.

Continuing the study of the sessions, we carve out the IP addresses mentioned in them. These are the addresses from which adversaries download their payloads. This amount of IP addresses is considerably smaller than the total amount of IPs that attacked the honeypot instances. However, 80 out of the 141 total IPs used for storage also attempted to open a Telnet session with one of the honeypot instances at some point. As adversaries are highly unlikely to allow their own infrastructure to be infected, it can be reasoned that storage servers are also used as the starting point of malware campaigns. Further, we perform WHOIS lookups on the unique IP addresses. All hosts behind IPs used to store payloads are property of, both small and big, hosting providers. The usage of hosting providers is to be expected. It allows for simple re-hosting in the case of a takedown and, in combination with fake identities, provides anonymity. Note that not all of these IPs were reachable at the time of analysis, confirming that adversaries frequently (have to) switch storage hosts. On the other hand, the majority of the thousands of remaining unique IP addresses that only attacked the honeypots, but are not known to host payloads, belong to telecom companies. In other words, they are actual infected hosts. Furthermore, as the main approach to intrusion is accessing the victim over Telnet, these hosts are likely to be IoT devices, rather than PCs and servers which do not tend to expose Telnet servers nowadays. The ones that still do belong to hosting providers are likely payload storage servers not accessed in any captured attack. The highly disproportionate ratio of storage servers to hosts matches the structure of botnets and validates our data. Meaning, the honeypot is believable to some extent. Lastly, notice that the amount of IPs used to host payloads is smaller than the total amount of collected unique binaries. This holds true even when compensating for architecture, as explained further on and seen in fig. 5.5. From this it can be concluded that adversaries distribute multiple variations of their malware. These variants are likely to include bug

fixes, updated wordlists, and new ways of exploitation.

Last but not least, the captured binaries are studied. The small ratio of unique binaries, based on their hash values, to the total amount of captured binaries implies that the honeypot instances have been attacked multiple times by the same malware samples. Note that changing even the smallest detail, such as the IP address of the C&C server which is commonly hardcoded in IoT malware, will affect the file's hash. Duplicate samples are thus, by definition, part of the same malware campaign. Using the network traffic captured with `tcpdump`, we are able to confirm that not only did remote IP addresses visit multiple times and deliver the same binaries on each visit, but duplicates were delivered by differing IP addresses as well. It can be concluded that victims are intentionally re-visited in order to ensure their infection. We argue the reason behind this to be twofold. First, it allows replacing a malware sample with the newest version. This ensures that the latest version is being used. A naive implementation of this "update" mechanism would simply always replace a prior malware instance on a victim, regardless of its version. As such, replacing it with the same version would be a frequent occurence. Second, persistency between reboots is hard to achieve for IoT malware. Being deleted from the filesystem, either by itself for evasion purposes or by the design of the device, means that a reboot results in complete removal of the malware. Furthermore, the overlap in unique binaries between the two instances means that both were infected with the same malware at some point. This explains how it is possible that peaks, i.e. attacks, on some days match between the two instances, as can be seen in fig. 5.2 and fig. 5.3.

We also look at the distribution of the captured unique binaries based on their target architecture. Uniqueness is determined based on the hashes of the samples. Figure 5.5 presents this distribution based on the architectures determined by the Unix `file` utility. As explained in section 4.1.1, the Telnet service runs in the emulation using the DCS-700L IP camera's firmware. This firmware uses the MIPS (32-bit LSB) architecture. Yet, 39.3% of all malware samples captured via Telnet are not compiled for this architecture. Meaning, they would not run on the target device. Although this is a significantly large percentage, ARM (32-bit LSB) on its own makes up almost half of this. It can thus be concluded that, most of the time, adversaries fingerprint the victim device's architecture and only download the appropriate binary.

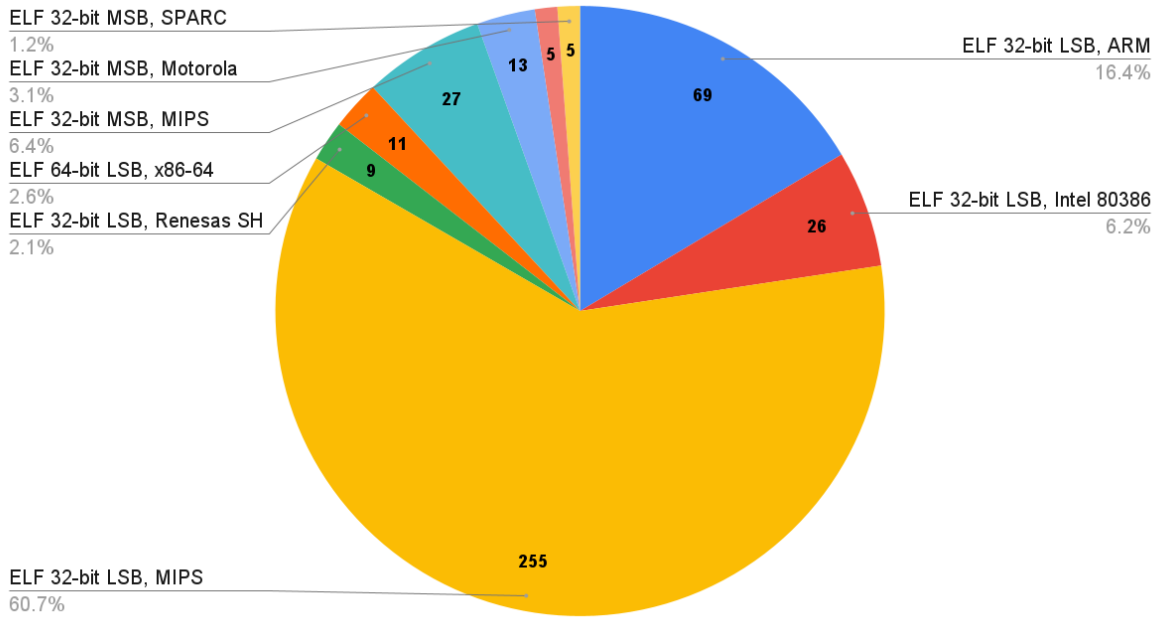


Figure 5.5: Distribution of captured unique binaries, based on their hashes, categorised by their target architecture.

5.2.2 Working of shell commands

In order to gain an insight in the actions performed after intrusion via Telnet, all sessions containing shell commands are clustered. First, all commands captured by the Telnet proxies are grouped per

session. Then, clustering algorithms require comparable quantifications, called feature vectors, for each input sample, i.e. session, to perform clustering. As to choose appropriate algorithms, we argue that sequences of shell commands can be compared with natural language sentences. Both consist of words combined within a given syntactic rule set. Note that a "word" in the context of shell commands is not fully equivalent to its definition in a natural language. Shell commands consist of names of commands and executables, strings, and special characters. In order to properly parse and split long sequences of instructions into "words", the `bashlex`[2] project is used. As for building feature vectors, this is done using the Term Frequency-Inverse Document Frequency (TF-IDF) approach as it is often used in Natural Language Processing (NLP) to compare documents. It builds a dictionary of all possible words and indicates per session the amount of occurrences of each word. Then, these values are divided by the total amount of occurrences of the respective words across all sessions. This compensates for the fact that certain words appear more frequently than others. As a result, more weight is given to the words that make the command sequences unique. The used clustering algorithm is K-means. Ten clusters are created. This number is determined experimentally by performing multiple runs with decreasing amounts of clusters. This was done until no clusters containing less than 10 sessions existed that had any obvious overlap with others.

Cleaning up of the data was required to achieve proper results. First, some commands have been logged incorrectly resulting in words not being space-separated. Second, several interactions contain less than a handful of words. We consider these to be sessions that ended prematurely due to unknown reasons. These sessions are not included in the clustering process. Lastly, as discovered in section 5.2.1, numerous remote IPs re-visited the honeypot instances. As such, many of the sessions are duplicates attempting to infect the instances with already seen malware. Only sessions with unique shell command sequences are used to compensate for this. Similarly, command sequences that are subsets of other sequences are also removed as these are likely the result of network failure, as discussed earlier. The resulting clusters are plotted in two dimensions in fig. 5.6. Information on each cluster is shown in table 5.3. The most influential words per cluster are listed in the table. These give an insight in the commands used in said clusters.

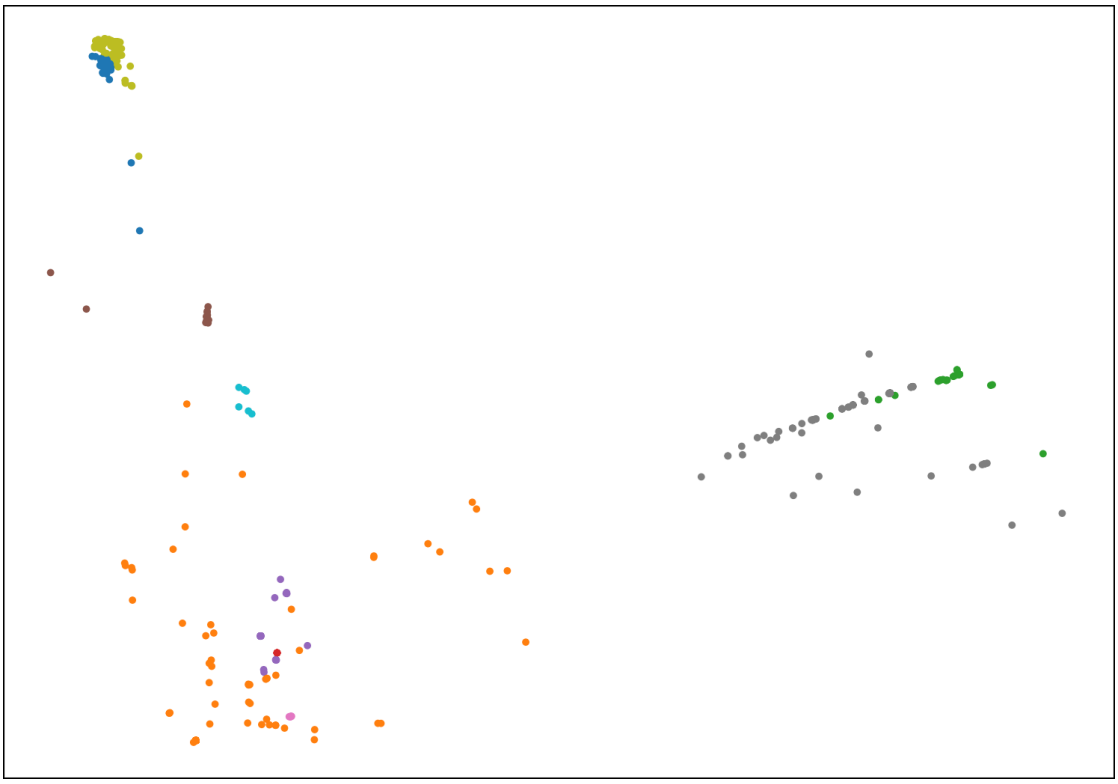


Figure 5.6: Graphical representation of the clustered sessions based on their associated commands. Only sessions with unique command sequences are included. Each color represents a different cluster, with each dot being a session.

[2] `https://github.com/idank/bashlex/`

Cluster	Unique sessions	Top 10 most influential words
1 (●)	611	/bin/busybox, sh, cd, -rf, rm, busybox, BOTNET, ||, +x, -en
2 (●)	130	/bin/busybox, -e, rm, >, cat, echo, ECCHI, /dev/.nippon, dvrHelper, /dev/pts/.nippon
3 (●)	91	/bin/busybox, -e, rm, cat, >, echo, /tmp/.none, /dev/.none, /home/admin/.none, /sys/.none
4 (●)	85	&&, >, /bin/busybox, cd, Iwsrfg, .06, nya, -rf, AA, wget
5 (●)	33	telnet, t.telnet, /bin/busybox, tftp, http://45.134.225.20/telnet, telnetd, 777, chmod, 45.134.225.20, sh
6 (●)	27	&&, /bin/busybox, >, hu87VhvQPz, cd, Switchblades1, ha7665caZS, -rf, ./hu87VhvQPz, /var
7 (●)	12	cat, echo, >, BOTNET, /bin/busybox, rm, /dev/.nippon, -e, /mnt/.nippon, /var/.nippon
8 (●)	7	whatttttttlol.sh, whatttttttlol1.sh, whatttttttlol2.sh, 23.94.50.159, http://23.94.50.159/whatttttttlol.sh, ||, anonymous, cd, sh, 777
9 (●)	6	/bin/busybox, if=/bin/busybox, bs=22, ||, count=1, dd, SATORI, while, do, <
10 (●)	5	/bin/ls, ||, adminpass, -n, ||, cd, $i, done, read, <

Table 5.3: Ranking of the clusters of sessions based on amount of associated sessions. Several words that influenced the clustering the most are shown per cluster, in decreasing order of importance. The colours match those used in fig. 5.6.

Despite best efforts to create unique clusters and using a relatively small number of them, fig. 5.6 still reveals several clusters related to each other. Similarities can be confirmed by comparing their lists of influential words. For example, the two clusters in the upper-left corner of the scatter plot, i.e. clusters with index 2 (light green, ●) and 3 (blue, ●), lay close to each other. And indeed, their first six words match. The reason for this can be determined by comparing any of the two clusters' sessions. Both clusters start and end with different shell commands. Yet, halfway through they perform the same actions in order to find a writeable directory. Other pairs of clusters found close to each other in fig. 5.6 can be compared similarly. In conclusion, malware implementations are closely related based on these similarities. This also explains the relatively small number of unique passwords as related implementations are likely to share wordlists. Antonakakis et al. confirms this idea as they state that various malware implementations and their wordlists evolved from Mirai's source code, which was made public by its alleged author [Ant+17; Kre16b].

We remark that all sessions successful at infecting the honeypots follow roughly the same pattern after intrusion of victim configuration, information gathering, dropping malware, executing it, and cleaning up. Manual analysis of samples from each cluster allows us to explain these steps in more detail. Following observations are made:

- Configuration: Vendors originally intend shell access for development and management purposes. To prevent unintended access, they might expose customised shells. These shells allow only certain commands to be executed. To escape this restricted environment, adversaries execute privileged actions, i.e. backdoors, or invoke traditional shells. This is such an important step that only 29 out of the 1365 unique sessions do not perform it. The others start with varying assortments of such commands. Examples are `enable`, `system`, `linuxshell`, `shell`, `config`, `terminal`, `start`, `telnetadmin`, `development`, `..`, `nc 1 1`, `sh`, and `bash`. Our honeypot implementation presents a traditional shell by default.
  Other, less frequently occuring, commands that adjust the environment are the following: `passwd`, `iptables -F`, and `mount -o remount,rw,exec <path>`. The first is used to change the current user's system password. Adversaries do this to lock out other malware, essentially claiming the device for themselves. Yet, their password of choice is `adminpass`. This itself is easy to guess and thus makes the practice of changing the password not all that effective. Although our honeypot implementation presents real firmware with a real `passwd` binary, changing the password requires entering the original user password. However, adversaries only have access to the least-privileged user created by us. Given that the password of this user is purposefully made to be complex and that the Telnet proxy handles authentication, adversaries are unable to actually go through

with any credential changes. The second command, `iptables -F`, flushes, i.e. deletes, all existing firewall rules. All further interactions, e.g. downloading the malware payload, talking to a C&C server, and starting and exposing custom services, are made possible by removing all networking restrictions. Again, this does not apply in our honeypot instances however. The reason being that the emulated environment does not contain any firewall settings. Data control is instead handled on the host, and consequently unaffected from commands performed by adversaries. Lastly, the `mount` command is used to change system permissions for storage devices mapped to certain paths. As mentioned before, vendors might mount parts of the filesystem as read-only. The above command attempts to remount a given path as both readable and writeable, and enables execution of files stored under it. While this command does technically work in the honeypots, all paths should be writeable by default resulting in this command to be a no-op.

- Information gathering: Just as the hardware between IoT devices differs greatly, so does the firmware. Malware must know varying pieces of information to be able to work in an automated manner. First, we discuss an alternative to the one discussed above for acquiring a filesystem path that is writeable. In practice, a completely read-only filesystem is inconvenient. For example, firmware updates must be downloaded or configurations must be written to files. As a result, certain paths are mounted as writeable by the firmware developers. Adversaries search for these paths in the following manner. Executing `cat /proc/mounts` lists all mounted paths. Then, a keyword along with the appropriate path is written to a file in each mount point's path. Depending on the result of reading back the files' contents, the adversary can determine which path can be used to store data.
  The second important piece of information is the CPU architecture of the victim device. In section 5.2.1 it was established that the captured binaries' architecture mostly match that of the victim. Although multiple variants of shell commands exist to get the information, they all work the same. Namely, by reading the header of the BusyBox binary that contains the architecture it is compiled for. This can be done by printing the binary to standard output with e.g. `cat /bin/busybox` or `while read i; do echo $i; done < /bin/busybox`. However, the binary data might corrupt the Telnet session. Thus, more preferable is to convert the binary to hex with e.g. `hexdump -e '16/1 "%c"' -n 32 /bin/busybox`. Third, some adversaries carve out only the ASCII text indicating the architecture from the header using `dd bs=52 count=1 if=/bin/busybox`. Last but not least, a handful of interactions include the listing of `/proc/cpuinfo`. This virtual file contains all CPU information including the architecture. Our high interaction implementation gives adversaries real binaries to work with, making this step work as intended for them.
  Lastly, several sessions are observed to read the contents of `/etc/passwd`, which is the list of all users on the system. The reason for this is unclear. However, all sessions doing this show similar command sequences and are thus of the same malware family.

- Dropping the payload: Two distinct approaches to transferring a malware binary onto the victim can be identified. These are downloading a payload over HTTP or FTP, and sending it over the already established Telnet connection. The latter is done by sending data encoded with the `uuencode` utility, hex encoded data, or base64 encoded data over Telnet, decoding it using an appropriate utility on the victim, and writing the resulting bytes to a file. Sometimes a shell script is fetched that in turn downloads multiple variants of the actual malware binary, each compiled for a different architecture. No fingerprinting is done when employing this latter method, which confirms the findings in section 5.2.1. Note that the act of sending encoded binaries instead of downloading them is highly uncommon. Only 4 out of the 255 unique MIPS (32-bit LSB) binaries, counted in fig. 5.5, are acquired this way. Lastly, occasionally a common Unix utility such as `telnetd` is removed and the malware binary is saved using its name. This helps to prevent detection as both the process and the file on the filesystem will look unsuspicious.

- Execution: Once the malware binary has been downloaded to the victim's filesystem, its permissions are adjusted to make it executable. After that, it is ran. No other notable shell commands are used in this step.

- Clean-up: As the name implies, in this step adversaries remove traces of the infection process or the malware itself from the filesystem. This includes clearing shell history, removing the files used to determine writeable paths, emptying or removing the payload file, and even removing all files under the current path. While commonly the last step, clean-up can also be performed at the start of a session. This prevents conflicts with already running malware instances. The approach is to

list all running processes and halt all recognised malware instances. Then, the old malware's file is removed from the filesystem. Out of the 1365 sessions with unique command sequences, 669 clean up to some extent. Furthermore, 261 out of these 669 check the running processes. However, as will be seen in section 5.2.3, clean-up functionality is sometimes included in the malware binary, removing the need to perform this step in the shell.

Inspection of command sequences also reveals techniques enabling automated exploitation. A new command can only be sent after the previous one has finished. Nevertheless, waiting for a specific line of output is not viable as most commands will output either nothing or non-deterministic text. The solution is to send two consecutive commands at once using the non-conditional separator `;`. The first command is the one with unknown output while the second one's output is static and thus known beforehand. For example, `echo` can be used to print a static string. However, a more prominent alternative is the command `/bin/busybox <string>`, where `<string>` can be any string except a valid BusyBox subcommand. Usually the string is random, the name of the malware, or the nickname of the person or group operating the malware. The error indicating an invalid BusyBox subcommand is static. It thus allows the adversary to know when the previous command has finished execution. The BusyBox approach is preferred as it has a second use case. That is, it allows one to confirm whether BusyBox is available on the system as the error message indicating its absence differs from that of the string not being recognised as a valid subcommand. This leads into the second technique enabling automation. Adversaries can not know beforehand what tools are available on the victim. Therefore, multiple possible alternatives are invoked and the first available one is chosen. This is done using the `||` separator. It only invokes the next command if the previous one returns an error code. A simple example is checking which base64 decoding utility is available using `/bin/busybox base64 --help || base64 --help || openssl base64 --help`.

Notably, no techniques to fingerprint honeypots could be identified with certainty in the recorded sessions. Nevertheless, we mention several commands that could be theorised to function as fingerprinting. First, the preference of using the BusyBox technique over the simpler `echo` one, to indicate finished commands, could be due to it also being used to detect low-interaction honeypots. Honeypots might not re-implement BusyBox properly, resulting in the invalid subcommand to not be handled appropriately. Second, several sessions execute `uname -r` and read the contents of the virtual file `/proc/sys/kernel/osrelease`. These should both return the same string, namely the kernel release. Although this can simply be a way to confirm whether a victim device is vulnerable, it could also be used to fingerprint low-interaction honeypots as, again, they might not implement both of these commands correctly. Lastly, as mentioned before, listing the contents of `/proc/cpuinfo` reveals the CPU architecture. However, it might also expose the fact that the CPU is a virtual device. In other words, it might expose that the victim environment is not running directly on physical hardware. The emulation of real firmware ensures yet again that these checks pass by returning valid data.

5.2.3 Malware samples

To study the workings of the collected malware samples, we construct an analysis pipeline. The pipeline first uploads a given binary to VirusTotal[3]. The VirusTotal web site provides, among others, scans by various antivirus services and IDSs, and community feedback. Then, the pipeline checks if the sample is packed with UPX and, if so, attempts to unpack it. In the next step, the malware binary undergoes static analysis. This consists of running it through YARA[4] with a collection of publicly available rules, `strings`, and ELF Parser[5]. These are respectively a signature-based scanner, a tool to extract human-readable data from blobs, and a tool that determines the capabilities of a binary, e.g. whether it reads files or creates network sockets. Last but not least, the malware samples are subjected to dynamic analysis by running them with QEMU for a set period of time. For this, we use the tools open sourced by the authors of [Alr+21]. Output logs, network logs, and traces of system calls are collected in this step. Only MIPS (32-bit LSB) binaries are analysed as the other captured samples differ solely in terms of architecture, as discussed in section 5.2.2: dropping the payload.

The VirusTotal and YARA steps of the pipeline show the capability of industry-leading tooling to identify the malicious samples. VirusTotal reports three-fourths of all samples to be Mirai, with the rest being identified as the Gafgyt malware. Similarly to Mirai, Gafgyt's source code has been made public over time [Lev16]. However, it is older than Mirai. Age is thus likely the reason for Mirai having a larger share

[3] `https://www.virustotal.com/`
[4] `https://github.com/VirusTotal/yara/`
[5] `https://elfparser.com/`

in the data. In terms of functionality, Mirai focuses on access by means of default credentials. Gafgyt, on the other hand, focuses on exploitation by means of abusing CVEs [Dav19]. Yet, even the earliest version includes a wordlist to access services with default credentials[6]. As both Mirai's and Gafgyt's source code is public, mixed variants have started to appear [Sea21]. This makes classification hard. In the rest of this section the malware families will be compared, instead of the honeypot instances. Table 5.4 lists statistics related to the preliminary steps of the pipeline.

Statistic	Mirai	Gafgyt	Total
Unique MIPS (32-bit LSB) binaries	192	63	255
New to VirusTotal	39	13	52
Median detection rate (%)	42.5	33.8	40.5
Unique CVEs detected	8	2	8
Median CVEs used	0	0	0
UPX packed	93	13	106
UPX unpacking failed	6	12	18

Table 5.4: Statistics on the captured MIPS (32-bit LSB) binaries in the preliminary stage of the analysis pipeline, categorised per malware family.

To our surprise, 20.4% of all captured MIPS (32-bit LSB) binaries did not have their hashes registered on VirusTotal prior to the analysis. This is based on the "first submission date" reported on the web site. The honeypot instances thus captured previously unknown malware samples. Given that there are only two big families, we assume this percentage to be so high due to the ease of creating variants. The ease of (re-)infection, due to lacking security measures, makes frequent adjustments to the malware code possible. An adversary is able to take the public source code of either malware type, apply their own customisations and/or improvements, and start a new malware campaign. Additionally, in section 5.2.1 we hypothesized that adversaries distribute updated versions of their malware. Despite influencing the hashes of the malware binaries that result from these changes, large portions of their code, and thus functionality, remain unchanged. Antivirus software can use these characteristics, among others, to determine which malware family the submitted binary belongs to. Furthermore, VirusTotal aggregates various antivirus services. Not each service can detect each malware with the same accuracy however. We call the ratio of services that detect a sample as malicious, to the services that do not, the detection rate. As can be seen in table 5.4, Gafgyt is harder to detect. The reason for this is unknown to us as it would require intrinsic knowledge of each antivirus service. Lastly, YARA was able to identify only eight unique CVEs with the given collection of rules. These are as follows: CVE-2014-2321, CVE-2014-8361, CVE-2016-10372, CVE-2017-17215, CVE-2017-18368, CVE-2017-18369, and CVE-2018-10561. However, only two CVEs were detected in the captured Gafgyt samples. This is remarkable given that its focus lies in abusing CVEs. We hypothesise the reason for this to be a combination of a small sample size, binaries failing to unpack, and insufficient coverage with our collection of YARA rules.

UPX packed binaries make up 41.6% of all MIPS (32-bit LSB) binaries. However, notice that 18 of these samples failed to unpack. Although the percentage of packed Gafgyt samples is smaller than that of Mirai, all but one of the packed Gafgyt samples failed to unpack. Further review shows that the binaries that fail to unpack still execute successfully in the dynamic analysis stage. Thus, it can be concluded that the authors tweaked some headers not essential to execution, as previously mentioned to be a possibility in section 2.4.2: evasion. Indeed, comparing the first few hundred bytes of a random binary packed by us with such a malware sample reveals a discrepancy between the two. A side by side comparison of the binaries' headers in hex format can be seen in fig. 5.7. Notice how the amount of bytes up till the second ELF magic bytes, marked in pink, is less in the malware binary (fig. 5.7b) than in the control binary (fig. 5.7a). Additionally, the malware binary does not contain the magic bytes `UPX!`. The malware author thus removed (parts of) the UPX header. In conclusion, the small amount of UPX packed samples that are also tweaked implies a low amount of effort to obfuscate IoT malware. This leads us to believe that the main purpose of using UPX is not to evade detection, but rather to decrease payload size. However, this only applies to the Mirai family. When the authors of a Gafgyt sample decide to pack their malware, they go the extra mile to make analysis harder. This is a surprising conclusion as Mirai is the younger malware family of the two. One would expect that adversaries gain experience over time, thus resulting in Mirai samples being more sophisticated than the older Gafgyt. We argue the opposite however. Mirai's

[6] `https://github.com/hammerzeit/BASHLITE/`

popularity attracted new, inexperienced adversaries, which further contributed to the malware family's growth. Yet, these new people do not possess the knowledge to protect their malware samples. They are also not required to gain said knowledge as the infection process does not necessitate it. On the other hand, the older, more experienced Gafgyt operators, whom wish to stay in business by occupying a niche, must better protect their binaries as defenders have learned and developed defensive measures over time.

```
00000000  7F 45 4C 46 02 01 01 03 00 00 00 00 00 00 00 00 02 00   .ELF..............
00000012  3E 00 01 00 00 00 E0 59 45 00 00 00 00 00 40 00 00 00   >......YE.....@...
00000024  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00   ................@.
00000036  38 00 03 00 40 00 00 00 00 00 01 00 00 00 05 00 00 00   8...@.............
00000048  00 00 00 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ..........@.......
0000005A  40 00 00 00 00 00 27 63 05 00 00 00 00 00 27 63 05 00   @.....'c......'c..
0000006C  00 00 00 00 00 10 00 00 00 00 00 00 01 00 00 00 06 00   ..................
0000007E  00 00 00 00 00 00 00 00 00 00 00 70 45 00 00 00 00 00   ...........pE.....
00000090  00 70 45 00 00 00 00 00 00 00 00 00 00 00 00 00 20 BC   .pE.............. .
000000A2  06 00 00 00 00 00 00 10 00 00 00 00 00 00 51 E5 74 64   ..............Q.td
000000B4  06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ..................
000000C6  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ..................
000000D8  00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 E2 3B   ................;
000000EA  19 90 55 50 58 21 50 09 0D 16 00 00 00 00 70 1F 0D 00   ..UPX!P.......p...
000000FC  70 1F 0D 00 70 02 00 00 CB 00 00 00 02 00 00 00 EE 7E   p...p............~
0000010E  90 FF 7F 45 4C 46 02 01 01 03 00 02 00 3E 00 01 07 90   ...ELF.......>....
00000120  15 40 86 7C 9B C5 0F 05 00 70 17 0D 00 60 C7 EE DE 13   .@.|.....p...`....
00000132  38 00 0A 05 20 00 1F 2B 04 00 00 4E C8 13 76 40 07 38   8... ..+...N..v@.8
00000144  05 00 00 10 D9 02 F6 CD 06 37 05 0F 07 40 81 25 E4 09   .........7...@.%..
00000156  25 C9 08 37 13 76 B0 B1 6F E0 17 E0 48 07 DB BB 03 4B   %..7.v..o...H....K
```

(a) The first few hundred bytes of a random binary packed by us. The UPX header is unaltered and contains the magic bytes `UPX!`.

```
00000000  7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00 02 00   .ELF..............
00000012  08 00 01 00 00 00 C8 47 10 00 34 00 00 00 00 00 00 00   .......G..4.......
00000024  07 10 00 00 34 00 20 00 02 00 28 00 00 00 00 00 01 00   ....4. ...(.......
00000036  00 00 00 00 00 00 00 00 10 00 00 00 10 00 55 51 00 00   ..............UQ..
00000048  55 51 00 00 05 00 00 00 00 00 01 00 01 00 00 00 6C AD   UQ..............l.
0000005A  00 00 6C AD 44 00 6C AD 44 00 00 00 00 00 00 00 00 00   ..l.D.l.D.........
0000006C  06 00 00 00 00 00 01 00 64 B9 9E 02 6D 2E 3B C3 B0 09   ........d...m.;...
0000007E  0D 1E 00 00 00 00 E0 AD 00 00 E0 AD 00 00 94 00 00 00   ..................
00000090  61 00 00 00 02 00 00 00 7F 3F 64 F9 7F 45 4C 46 01 00   a........?d..ELF..
000000A2  02 00 08 00 0D 60 02 40 7F 2B B7 DD 00 34 07 B0 AB 03   .....`.@.+...4....
000000B4  07 10 0B 20 00 03 00 28 00 23 8F 5D F6 0E 00 0D 00 2D   ... ...(.#.]....-
000000C6  40 03 E0 A5 00 05 46 BE 63 63 13 01 1F E4 A5 03 44 68   @.....F.cc......Dh
000000D8  D8 0F B6 ED 18 88 07 03 06 1F 51 E5 74 64 00 00 00 C0   ..........Q.td....
000000EA  ED 02 22 00 04 03 00 00 00 00 00 00 00 90 FF 4C A5 00   ..".............L..
000000FC  00 05 44 00 00 02 00 00 00 BB FF FF FF 05 00 1C 3C 8C   ..D.............<.
0000010E  27 9C 27 21 E0 99 03 E0 FF BD 27 10 00 BC AF 1C 00 BF   '.'!......'.......
00000120  AF 18 07 01 00 EF 8F 7C EE 11 04 00 00 23 68 9F 03 20   .......|.....#h..
00000132  80 99 8F 13 B0 98 7B FF DC 01 39 27 09 F8 20 03 0B 33   ......{...9'.. ..3
00000144  13 2F C9 41 06 39 38 1C B0 A0 DA F6 98 5B 67 07 08 00   ./.A.98......[g...
00000156  E0 57 00 7B 9B 6B 06 FB 37 00 8B D8 20 87 23 B1 DF C7   .W.{.k..7... .#...
```

(b) The first few hundred bytes of a malware sample that failed to unpack. The UPX header is altered as the amount of bytes between the two sets of ELF magic bytes is small, and the magic bytes `UPX!` are not present.

Figure 5.7: Comparison of a packed control (top) and malware (bottom) binary. Corresponding ELF magic bytes are marked in pink.

Next, we look at the capabilities of the malware samples. The ELF Parser utility uses several heuristics to determine what possible actions an ELF executable can perform. Figure 5.8 shows a measurement of the detected capabilities. A skewed ratio between Mirai and Gafgyt binaries is expected as the amount of captured samples differs between them. As such, comparing them in absolute numbers is not fitting. We instead focus on explaining what each capability entails, exemplifying where appropriate, and how they tie into the general workings and goal of IoT malware by combining this data with the results of the dynamic analysis phase.

The popularity of shell commands contained within the samples comes as no surprise given that they propagate by sending commands. The strings labelled as HTTP functionality give more insight however. First, various HTTP user agent strings, both of well-known software and random strings, are detected. Using a different user agent on every request hinders detection. Furthermore, presenting a user agent used by other software prevents being blocked based on this value. Other strings classified under HTTP

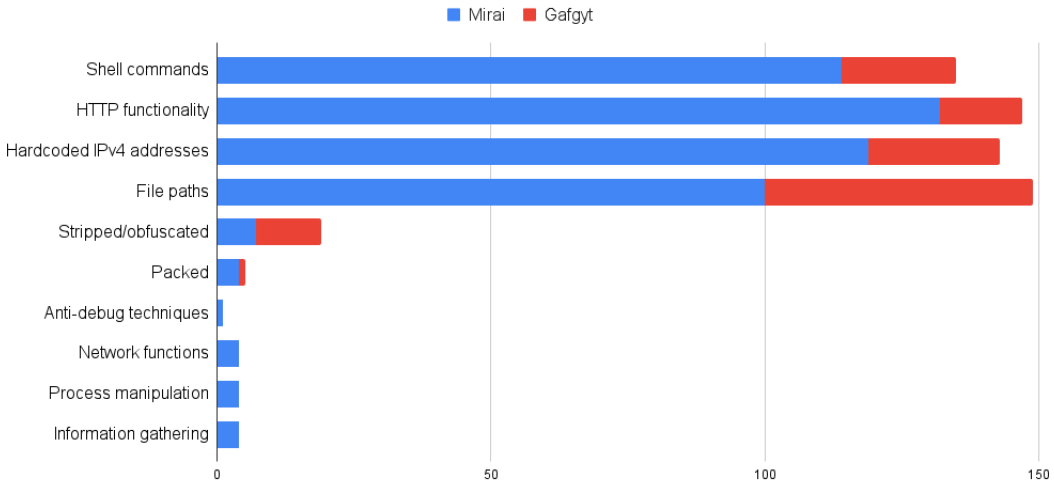


Figure 5.8: Capabilities of captured binaries, per malware family. The total number of analysed, unique MIPS (32-bit LSB) binaries is 255.

functionality are paths and XML payloads. We identify these to be parts of HTTP and UPnP SOAP requests that exploit known vulnerabilities. Manual review reveals at least six more vulnerabilities not detected by the YARA rules. These are the following: RCE in Linksys E-series devices [Ull14], RCE in TVT Shenzhen DVRs [Kim20], CVE-2019-9082, CVE-2020-8958, CVE-2020-10173, and CVE-2021-44228. CVE-2021-44228 is more commonly known as Log4Shell, an arbitrary command execution vulnerability in the popular Java logging library Log4j. The library's prevalence and relative ease of exploitation quickly made the weakness notorious. Including the CVEs mentioned earlier, the samples can exploit weaknesses spanning almost a decade. However, age is not an issue for IoT malware as the target devices are rarely updated due to a lack of human supervision. Although old, the vulnerabilities share certain characteristics. Namely that they are easy to exploit and have high impact. More specifically, they are unauthenticated shell command injection vulnerabilities. Combined with the poor permission and user management of IoT devices, adversaries are able to gain root privileged shell access without much effort by abusing these vulnerabilities. As for hardcoded IP addresses, 61 unique public addresses are identified. Roughly half of them overlap with the IP addresses used in Telnet sessions to download payloads, as described in section 5.2.1. The other half consists of either unidentified storage hosts or C&C servers. WHOIS information of the latter half reveals that they belong to hosting providers, supporting the possible use cases. Furthermore, three unique addresses belonging to private IP address ranges are contained in the binaries. All three are part of larger strings that are components of exploits mentioned above. How these are used is explored further in section 5.5.

The hardcoded paths also confirm previously discussed behaviour. Identified paths can be grouped into three groups:

- Payload file path: This group consists of paths such as `/tmp/.e` and `/tmp/aqua.mpsl`. These are hardcoded paths to which the malware will write itself when infecting another device. Dynamic analysis shows that the malware sample's location on the filesystem can also be used to remove itself after having started in order to leave no traces.
- System configuration files: These are file paths such as `/etc/resolv.conf`, `/etc/services`, and `/proc/cpuinfo`. These files were not accessed during the dynamic analysis phase, meaning that they are to be used during interactions with victims. The files can be read to gather information on how to proceed with intrusion, as well as check whether the victim environment is a real device. The capability of information gathering lists these paths as well.
- System tooling: This group contains e.g. `/usr/bin/echo`, `/dev/watchdog`, and `/usr/lib/systemd`. We choose these examples as they match three possible actions to be undertaken using system tooling:

- First, when intruding on a device, the malware will use already existing system tooling to prepare and infect the device. This has been discussed in section 5.2.2. The malware binary needs these strings to be able to send them to the victim.

- Second, in section 2.4.2 it was mentioned that malware attempts to gain persistence by preventing a reboot. This requires fooling the watchdog of the OS. The Linux watchdog works as follows [WO07]. A service in user space of the OS is responsible for periodically notifying the kernel watchdog. This kernel watchdog in turn notifies a hardware watchdog. If everything works as intended, no timeout occurs and no hard reset happens. Unresponsiveness of the user space service will result in missing notifications and thus a reboot. This may be caused by any kind of system failure including lack of resources or the halting of the user space service, which can both be caused by the malware binary. The dynamic analysis reveals how this is solved in practice by malware authors. Namely, they create their own service that notifies the kernel watchdog. Doing so is as simple as forking a thread that periodically writes random data to the watchdog file. However, note that `/dev/watchdog` is not the only possible location, and name, for this file. `/etc/watchdog`, `/dev/misc/watchdog`, and `/dev/FTWDT101_watchdog` are only some of the alternatives checked for by the captured malware samples. Vendors are free to customise their firmware and rename this file as needed. As a result, malware authors must include various paths in an attempt to find the watchdog file. If finding and writing to it is unsuccessful, they can either ignore the functionality or halt the infection completely. Remember, as described in section 4.2.3, our implementation gives intruders no root permissions. Yet, writing to the watchdog file requires said permissions. Although the intrusion and binary would already be logged by the honeypot at this point, it nevertheless hampers the believability of our implementation.

- After having infected a device, malware will try to hide itself on the victim to evade detection as discussed in section 2.4.2: evasion. The third type of paths to system tooling contains tools that are not used during intrusion, but are well-known utilities nevertheless. Malware can fool human and naive automated detection tools by removing such utility from the victim and replacing it with itself.

Continuing with evasion techniques, we discuss the capability of process manipulation. Using the system call `prctl`, malware samples rename their reported process' name. Furthermore, dynamic analysis reveals the scanning of all running processes by reading files in the `/proc` directory. Processes with the same name as the malware are halted. This enables re-infection by preventing duplicate instances of malware to be ran simultaneously. Indeed, several samples contain the string `[single-instance] killing other bot: %s - pid: %s`, confirming that duplicate instances are undesired. Despite not observing it in our test setup performing dynamic analysis, IoT malware is also known to scan and halt competing malware [Voo+19]. Manual review of all human-readable strings in the captured binaries reveals one sample that contains a list of more than 300 competing malware names. Note that, as explained above, preventing access to the `root` user might affect interactions with the watchdog. However, the decision to do so also allows us to hide processes of the emulation framework from the least-privileged user. This is important here as it future-proofs the implementation from being detected by malware.

The packed capability indicates binaries packed with UPX. However, not all samples are detected by the ELF Parser utility due to modified UPX headers. Further, anti-debug techniques pertains to stripped or obfuscated ELF headers. Upon further inspection of samples with this capability, we discover that some adversaries have implemented their own simple packer. The entry point of the malware loads a chunk of bytes from a different part of the binary, performs some decoding operations, and then directs execution to it. Storing instructions in such obfuscated way complicated malware analysis.

Last but not least, the capability of network functions, as expected, contains functions to handle network sockets. From the functions to open, send, receive, and close sockets, the `setsockopt` used to configure socket options stands out. An example invocation captured during dynamic analysis is `setsockopt(3<RAW:[1073]>, SOL_IP, IP_HDRINCL, [1], 4)`. This applies the `IP_HDRINCL` option to the socket. It allows the malware binary to tweak the IP header of packets it sends instead of using the header created by the kernel. The use case for this is spoofing the source IP address either during network scanning, as to prevent being blocked, or when performing e.g. SSDP reflection attacks.

5.3 HTTP usage

To gain an understanding of the captured web traffic, all HTTP requests are first classified into sequences. A sequence is a directed graph and represents the browsing behaviour of a client. Each node is a web resource fetched by the client, and each edge indicates the order in which the nodes, i.e. resources, are visited. An edge can be determined based on the resource that lead the client to the current resource, i.e. using the `Referer` HTTP header. Sequences may also branch when a web page automatically fetches resources. They illustrate the browsing behaviour of clients and are thus fitting representations due to the web's linked structure. Then, all sequences are grouped together by merging nodes that have the same URL, same depth, and matching ancestry. This creates a directed graph with the shape of a tree that shows all possible paths taken by clients.

To start, incoming requests will be analysed. Figure 5.9 shows the merged sequences for incoming requests. Several limitations apply to the figure to retain clarity. Due to the exponential nature of branching of the graph, only two levels of depth are shown, i.e. the first and second requested resource when visiting. For the same reason, resources at a depth of two are only displayed if they have been visited a minimum number of times. Additionally, only the destination path is shown when the target host is one of the honeypot instances. Several observations can be made about the behaviour of clients when they visit our honeypot instances using this graph.

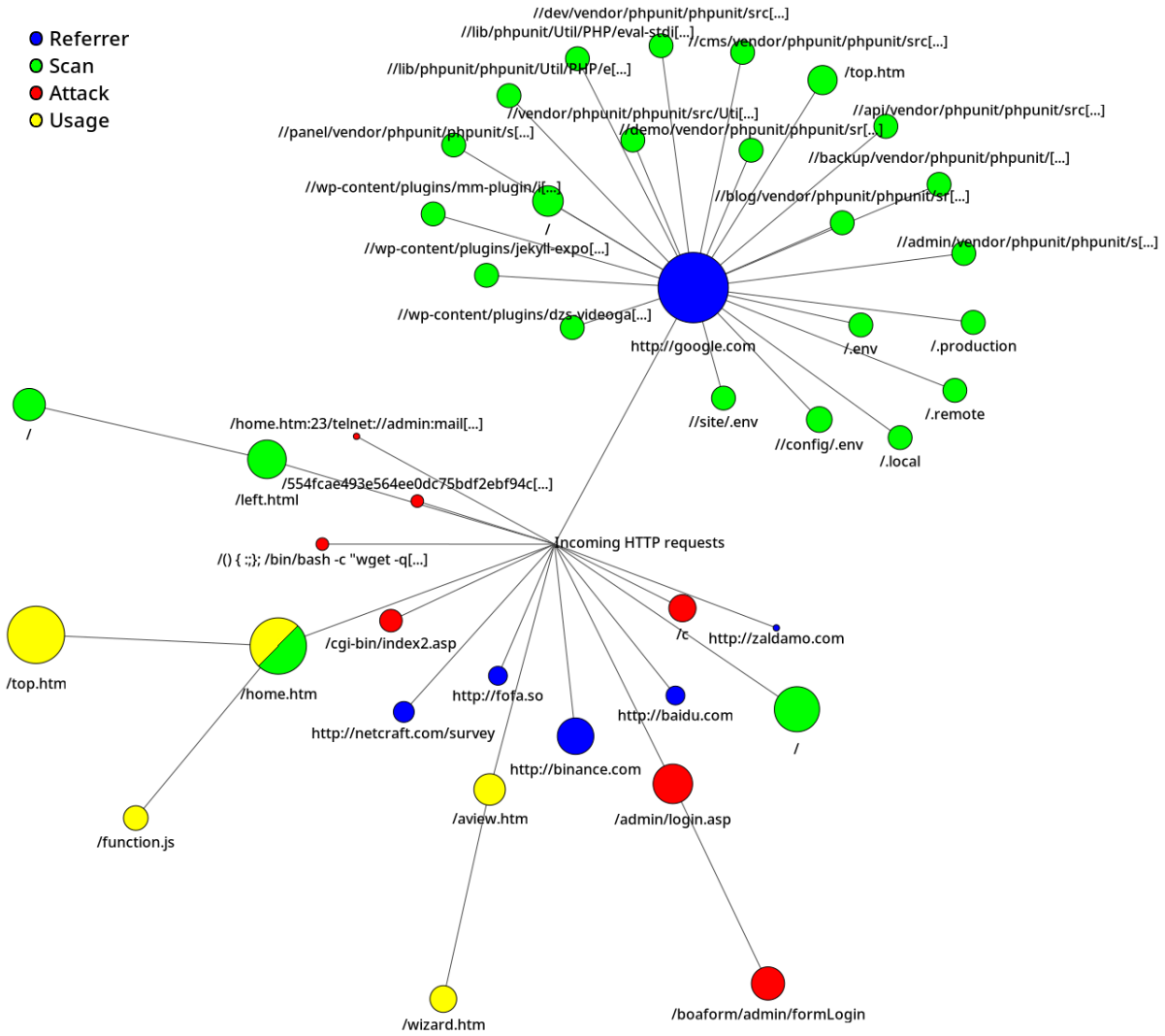


Figure 5.9: The first and second step in the HTTP sequences of clients visiting the honeypot instances' web services. Only the destination path is shown when the target host is a honeypot instance. The color of each circle indicates the reason for the request, while the size is proportional to the amount of requests made. Long URLs are cut off and indicated with [. . .].

First, notice that clients supposedly start their browsing on popular web sites such as `https://baidu.com`, `https://binance.com`, and `https://google.com`. The second step in their sequences reveals their intention however. The children of e.g. the `https://google.com` node point to common paths that are almost all not available on the honeypot's web server. Despite the size of the parent node, all child nodes have individually been visited a small number of times. The number of visits is also roughly the same for all child nodes. It can be concluded that these visits are the result of automated scanners. For these scans, adversaries manually set the `Referer` header to a popular web site. We theorize that by doing this, the traffic is made to look like a human happened to stumble upon the victim through the parent node's web site. This might confuse naive defensive systems. Such scanners are able to generate a substantial amount of requests. This explains at least some of the peaks of incoming HTTP traffic in fig. 5.2.

Second, other requests attempt to exploit vulnerabilities in the web server of the IoT devices. As established in section 5.2.3, aside from intruding on Telnet with default credentials, IoT malware also attempts to exploit well-known vulnerabilities in the firmware of devices. For example, the most visited path with the intent of attacking a honeypot instance is `/boaform/admin/formLogin`, which is disguised to be coming from `/admin/login.asp` with the `Referer` HTTP header trick. This path is the target of CVE-2020-8958, which we earlier identified to be used by several of the captured malware binaries. In other words, these requests are a confirmation of our prior observation that IoT malware employs CVEs to infect victims.

Third are the requests made to resources hosted on the honeypot instances. As discussed in section 4.1.1, our implementation hosts a web administration panel for an IP camera. Accessing it requires authenticating with the username and password `admin`. Table 5.5 lists the three stages of interacting with the web administration panel. As mentioned in the introduction to this chapter, logs of the AWS instance were lost between certain periods of time. Due to a big discrepancy in the amount of captured visits, we also show the statistics of the OVH instance after compensating for said period. The concrete reason for the discrepancy is unknown. However, we hypothesise following properties to be influencing this phenomenon. As mentioned before in section 4.3.4, the hosting provider may impact the audience that targets each instance and how they behave. Next, both honeypots are installed on hosts that received a clean OS install beforehand. Yet, while the AWS instance along with its IP address have been acquired new, the OVH instance has previously been used for a different research project. Not only did its IP address remain unchanged, a DNS A record that points to it also still exists at the time of analysis. None of the clients include said DNS record in the `Referer` HTTP header however. This indicates that they did not land on the web page of the honeypot through existing links. Thus, a more likely explanation is that the age of the IP address and the DNS record improve the believability of the host, and consequently the honeypot instance hosted on it. The Internet scanning service Shodan[7] provides an API that attempts to identify whether a host behind an IP address is a honeypot or not. Remarkably, the AWS instance is, with certainty, marked as a honeypot while the OVH instance is not. This not only supports our hypothesis, but it might be increasing the legitimacy of the OVH host even further. This information can be used for future implementations. It either encourages honeypot operators to ensure no hosts are reused from prior work, or they might deliberately use older IP addresses and/or DNS records to achieve better results. Nevertheless, the two should not be mixed to prevent harming the integrity of the captured data, as happened in this thesis.

Statistic	AWS EC2	OVH VPS (comparable)	OVH VPS (full)
Visit `/home.htm`	5	37	177
Successful login	0	28	119
Continue to browse	0	8	16
Median browsing duration (seconds)	/	54.6	57.6
Shortest browsing duration (seconds)	/	20.2	20.2
Longest browsing duration (seconds)	/	94.1	236.4

Table 5.5: Statistics on the three stages of interacting with the web administration panel honeypot service, categorised per instance. A / means that no applicable value exists.

We continue analysis on all interactions due to the small sample sizes. Only roughly two-third of all clients manage to properly authenticate against the IP camera's administration panel, as can be seen in

[7] `https://honeyscore.shodan.io/`

table 5.5. An even smaller fraction of clients go on to actually interact with the web site. Comparing the duration of the interactions with those of Telnet sessions, discussed in section 5.2.1, it can be concluded that these are performed by humans. Unfortunately, none of the 16 browsing client's IP addresses appear in the Telnet proxy's logs. This means that they did not use any honeytoken. We thus can not know how they came to find the honeypot instance. Although each visitor shows interest in the camera feed, watching it for several seconds, the general behaviour of clients can be divided into three groups:

- The first group only views the camera feed and leaves relatively fast.
- The second group is more interested in the camera feed. They attempt to interact with it by e.g. toggling night mode or the audio on/off. Yet, as explained in section 4.1.1, the emulation of the IP camera's firmware is not perfect. As a result, we had to implement a patch to be able to show a camera feed. Unfortunately, the patch turned out to be too simplistic as the said clients left right after observing no changes resulting from their actions.
- The last group is less interested in the camera feed, and decides to browse to the settings and system statistics pages. However, they do not change any settings, i.e. interact with these pages.

Thus, when a human happens to login to an exposed IP camera, they will be drawn to the camera feed. Some might explore further, but none are eager to start changing any settings. Note that this conclusion is limited by both the small sample size and the mistake of reusing an older host with a registered DNS record, as discussed above. Furthermore, the latter in combination with the lack of knowledge as to how the clients found the honeypot means that it can not be assumed that the clients are visiting with malicious intent. This could also explain why none of them proceeded to change any settings. What can be said with certainty, however, is that a proper patch for the camera feed would have benefited the honeypot implementation. More detailed logs can be gathered if visitors are kept engaged for longer. Future work must ensure high fidelity on all fronts of the implementation.

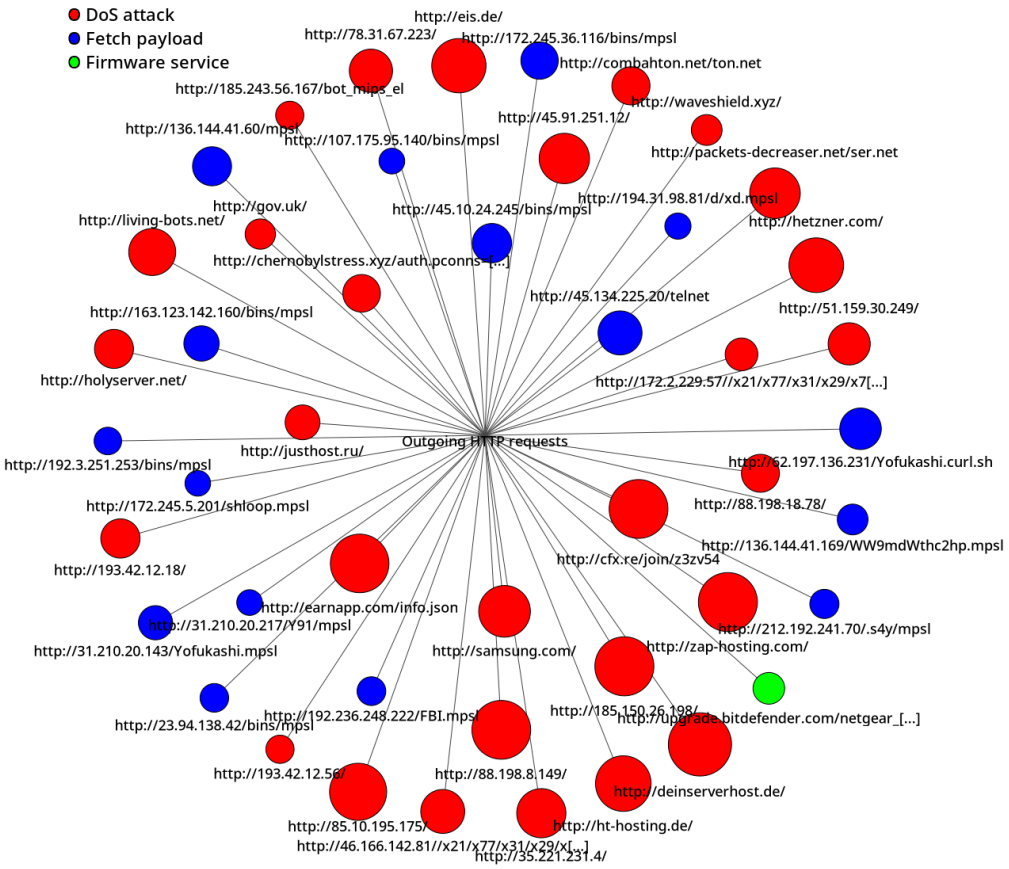


Figure 5.10: Hosts that have been visited at least a hundred times by the honeypot instances over the analysed time period. The color of each circle indicates the reason for the request, while the size is proportional to the amount of requests made. Long URLs are cut off and indicated with [. . .].

Last but not least, we also have a look at outgoing HTTP requests. Figure 5.10 shows URLs that have been visited at least a hundred times by the honeypot instances. This limitation must be applied for clarity of the figure. It prunes hundreds of nodes that have been visited only a handful of times. In the figure, several groups can be identified. First are the endpoints that were targeted during DoS attacks. Shortly after becoming infected by a malware sample, a honeypot instance would receive a TCP packet from a supposed C&C server. Using the options in said packet, the honeypot would then participate in the attack by sending numerous HTTP requests. The start of an attack targeting `https://gov.uk` is shown in fig. 5.11. A large amount of HTTP requests is matched by a similar amount of HTTP responses from the victim. This explains some of the peaks of incoming HTTP traffic, such as the one that occurred on the OVH instance between the 10th and 17th March, 2022. As a reminder, the firewall described in section 4.2.4 severely limits outgoing bandwidth. As such, the contribution of our instances to such attacks is negligible. Nevertheless, it shows that our implementation is not only capable of hosting firmware with a fidelity that allows infection and even usage of the malware, but also allows recording all related traffic.

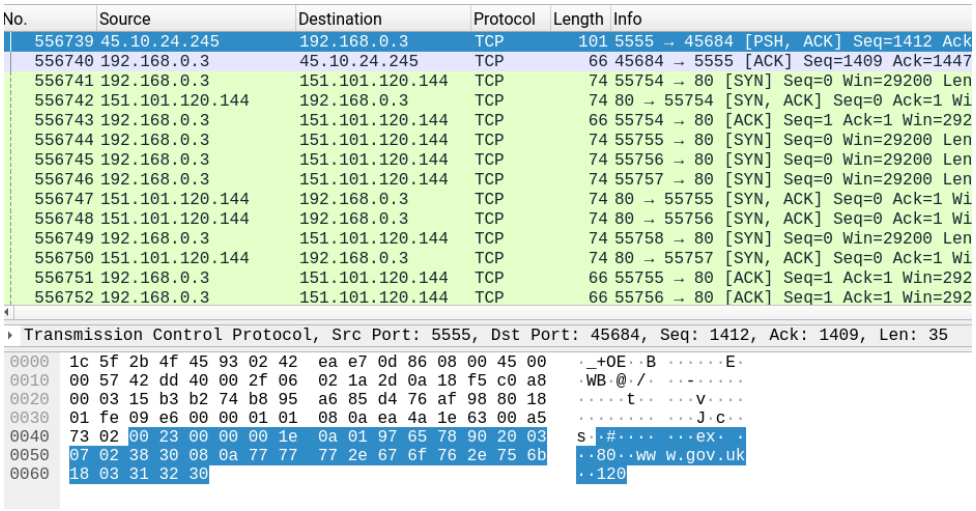


Figure 5.11: Excerpt from the captured network logs showing the start of a DoS attack. A TCP packet is received containing the strings 80, www.gov.uk, and 120. These are the port, URL, and extent of attack options respectively. Several requests targeting the victim follow.

The second group of visited endpoints consists of the ones used to download malicious payloads, i.e. malware samples. These can mainly be identified by the paths being requested. Speaking of payloads, the third group consists of a single URL, namely `http://upgrade.bitdefender.com/netgear_r7000p_boxlight/versions3.id`. This was being visited by the router honeypot service, which emulates the firmware of a Netgear® R7000. The reason for these visits is that the router attempts to fetch automatic updates to one of its services. This brings up a downside of emulating consumer firmware in a high interaction honeypot. Consumer firmware is configured by the vendor to automatically start services that will attempt to contact outside resources. The honeypot's operators must be aware of all such services included in the base firmware and handle related data appropriately. This removes possible confusion and misinterpretation of results. Despite filtering out the data related to vendor services during the analysis phase, we did not consider the existence of such services during the implementation of our honeypot. This is a necessary improvement in future work.

We also have a look at the endpoints that have been visited less than a hundred times, and are thus not shown in fig. 5.10. Aside from containing IP addresses hosting malware samples that have been fetched less frequently, two more groups can be identified within the data. Both groups are related to the workings of the malware samples that infected the honeypot instances. The first group consists of APIs that respond with the client's public IP address. This is part of the information gathering step of malware binaries. The public IP address of the infected victim is reported to the adversary's C&C server such that commands can be issued at a later date. The second group contains attempts at exploiting other IoT devices' web services, similar to the captured exploits and the ones discussed in section 5.2.3.

5.4 UPnP/SSDP abuse

The UPnP protocol has been designed for M2M communication. The SOAP API presented by UPnP services enforces this philosophy. This API informs control points of the device's capabilities and provides a structure to follow during further communication. It not only removes the need for automated scanning, but ensures that possible interactions are limited and well-defined. This, in turn, facilitates the analysis of the traffic hitting a UPnP service. We can deduce how adversaries intend to misuse the exposed service by looking at incoming traffic that attempts to invoke a response from it. The logs show only a handful of unique interactions. These are, aside from harmlessly listing the possible actions of the device, M-SEARCH and POST requests to interact with the SSDP and SOAP parts of the UPnP service respectively.

No.	Source	Destination	Protocol	Length	Info
484383	47.94.165.165	192.168.0.2	SSDP	132	M-SEARCH * HTTP/1.1
484384	47.94.165.165	192.168.0.2	SSDP	132	M-SEARCH * HTTP/1.1
484385	192.168.0.2	47.94.165.165	SSDP	354	HTTP/1.1 200 OK
484386	47.94.165.165	192.168.0.2	SSDP	132	M-SEARCH * HTTP/1.1
484387	192.168.0.2	47.94.165.165	SSDP	356	HTTP/1.1 200 OK
484388	47.94.165.165	192.168.0.2	SSDP	132	M-SEARCH * HTTP/1.1
484389	192.168.0.2	47.94.165.165	SSDP	292	HTTP/1.1 200 OK
484390	47.94.165.165	192.168.0.2	SSDP	132	M-SEARCH * HTTP/1.1
484391	47.94.165.165	192.168.0.2	SSDP	132	M-SEARCH * HTTP/1.1
484392	192.168.0.2	47.94.165.165	SSDP	301	HTTP/1.1 200 OK
484393	192.168.0.2	47.94.165.165	SSDP	364	HTTP/1.1 200 OK
484394	192.168.0.2	47.94.165.165	SSDP	301	HTTP/1.1 200 OK
484395	192.168.0.2	47.94.165.165	SSDP	340	HTTP/1.1 200 OK
484396	192.168.0.2	47.94.165.165	SSDP	301	HTTP/1.1 200 OK
484397	192.168.0.2	47.94.165.165	SSDP	360	HTTP/1.1 200 OK
484398	192.168.0.2	47.94.165.165	SSDP	356	HTTP/1.1 200 OK
484399	192.168.0.2	47.94.165.165	SSDP	372	HTTP/1.1 200 OK
484400	192.168.0.2	47.94.165.165	SSDP	366	HTTP/1.1 200 OK
484401	192.168.0.2	47.94.165.165	SSDP	354	HTTP/1.1 200 OK
484402	192.168.0.2	47.94.165.165	SSDP	356	HTTP/1.1 200 OK
484403	192.168.0.2	47.94.165.165	SSDP	292	HTTP/1.1 200 OK
484404	192.168.0.2	47.94.165.165	SSDP	301	HTTP/1.1 200 OK
484405	192.168.0.2	47.94.165.165	SSDP	364	HTTP/1.1 200 OK
484406	192.168.0.2	47.94.165.165	SSDP	301	HTTP/1.1 200 OK
484407	192.168.0.2	47.94.165.165	SSDP	340	HTTP/1.1 200 OK
484408	192.168.0.2	47.94.165.165	SSDP	301	HTTP/1.1 200 OK
484409	192.168.0.2	47.94.165.165	SSDP	360	HTTP/1.1 200 OK
484410	192.168.0.2	47.94.165.165	SSDP	356	HTTP/1.1 200 OK

```
▸ Internet Protocol Version 4, Src: 47.94.165.165, Dst: 192.168.0.2
▸ User Datagram Protocol, Src Port: 4859, Dst Port: 1900
▾ Simple Service Discovery Protocol
  ▾ M-SEARCH * HTTP/1.1\r\n
    ▸ [Expert Info (Chat/Sequence): M-SEARCH * HTTP/1.1\r\n]
      Request Method: M-SEARCH
      Request URI: *
      Request Version: HTTP/1.1
    Host:239.255.255.250:1900\r\n
    ST:ssdp:all\r\n
    Man:"ssdp:discover"\r\n
    MX:3\r\n
    \r\n
```

Figure 5.12: Excerpt from the captured network logs showing an SSDP reflection attack in the works.

As described in section 2.5.2, the SSDP component of UPnP can be abused in a reflection attack. Figure 5.12 shows an example of one of the honeypot instances being misused for this purpose. The attacker spoofed their IP address to be that of the victim, i.e. `47.94.165.165`, and sent thousands of M-SEARCH requests to the honeypot's UPnP service. The Search Target (`ST`) field is set to `ssdp:all`, which requests all services that the device has to offer. The UPnP service in turn sends a `200 OK` response per available service to the victim. The Netgear® R7000's firmware, of which the honeypot UPnP service is part of, provides 41 services. Figure 5.13 shows a direct comparison of the amount of incoming to outgoing SSDP frames over time. A log scale is used to retain clarity despite the occasional high peaks of traffic. As can be seen, the amplification is between one or two factors of ten on the average day. Yet, when the graph peaks, the difference between the amount of incoming and outgoing frames decreases. The reason for this is twofold. First, the emulation is unable process, and keep up with, the amount of requests. Second, the honeypot implementation uses an abuse checking service, discussed in section 4.2.4. A short period after detecting abuse, here identified by unusually high amounts of outgoing traffic, it clean resets the honeypot instance. This results in numerous SSDP requests that end up not being answered. Neverthe-

less, as can be seen in fig. 5.12, the size of an M-SEARCH frame is only between half to a third of the size of a response. Thus, it is evident that this attack is simple and capable of generating large amounts of traffic.

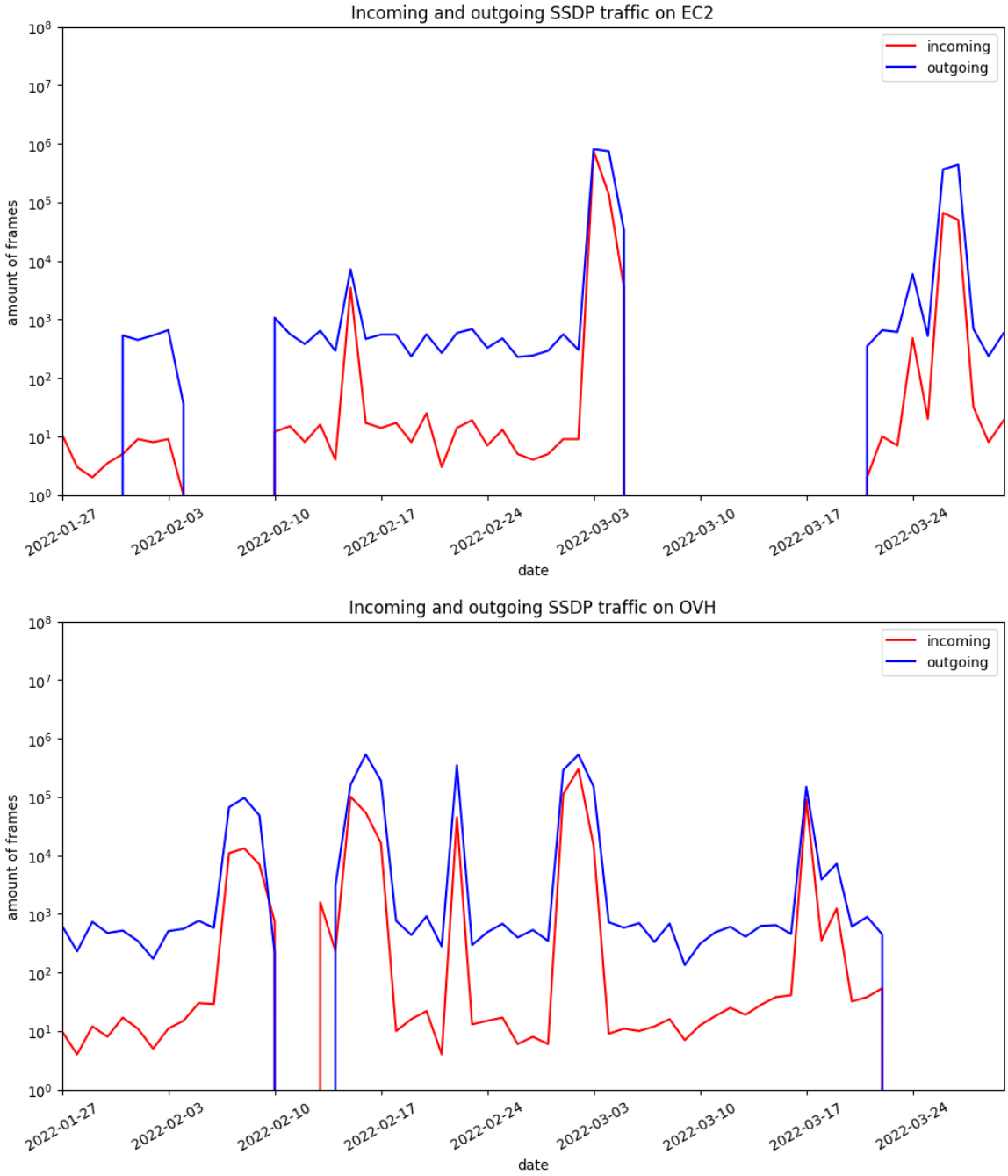


Figure 5.13: Direct comparison of incoming and outgoing SSDP traffic, per instance, over the analysed duration.

The SOAP APIs of the honeypots were also targeted. The behaviour of the adversaries, of which there are only two distinct ones, is peculiar however. The used IP addresses do not belong to publicly known scanning services and are thus hosts that are being abused. Furthermore, they do not appear in any other service's captured logs. Both would first request `/Public_UPNP_gatedesc.xml`, i.e. the root path describing the possible actions for an IGD. This is likely in order to confirm whether the discovered port is indeed exposing an IGD's UPnP service. After that, they would attempt to interact a single time with the device. This would repeat for the whole duration of the honeypots being online. Each adversary would send the same, although different from each other, requests each time. Examples of the two interactions are shown in fig. 5.14. It can be seen that both requests are answered with an error.

```
POST /Public_UPNP_gatedesc.xml HTTP/1.1
Host: 51.91.9.167:5000
Accept-Encoding: identity
Content-Length: 338
SOAPAction: "urn:schemas-upnp-org:service:WANIPConnection:1#GetGenericPortMappingEntry"
Content-Type: text/xml

<?xml version="1.0" ?><s:Envelope s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"><s:Body><u:GetGenericPortMappingEntry
xmlns:u="urn:schemas-upnp-org:service:WANIPConnection:1"><NewPortMappingIndex>0</
NewPortMappingIndex></u:GetGenericPortMappingEntry></s:Body></s:Envelope>HTTP/1.0 404 Not
Found
Content-type: text/html

<html>
<head><title>404 Not Found</title></head>
<body><h1>404 Not Found</h1>
<p>The resource you have requested is not available.</p></body>
</html>
```

(a) An attempt to fetch the first entry in the port mapping table. However, it was sent to the wrong path.

```
POST /Public_UPNP_C3 HTTP/1.0
HOST: 192.168.0.2:5000
SOAPACTION:"urn:schemas-upnp-org:service:WANIPConnection:1#AddPortMapping"
CONTENT-TYPE: text/xml ; charset="utf-8"
Content-Length: 637

<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
<u:AddPortMapping xmlns:u="urn:schemas-upnp-org:service:WANIPConnection:1">
<NewRemoteHost></NewRemoteHost>
<NewExternalPort>25473</NewExternalPort>
<NewProtocol>TCP</NewProtocol>
<NewInternalPort>80</NewInternalPort>
<NewInternalClient>172.217.20.78</NewInternalClient>
<NewEnabled>1</NewEnabled>
<NewPortMappingDescription>sync-25473</NewPortMappingDescription>
<NewLeaseDuration>600</NewLeaseDuration>
</u:AddPortMapping>
</s:Body>
</s:Envelope>
HTTP/1.1 500 Internal Server Error
EXT:
CONTENT-TYPE:text/xml
SERVER:Linux/2.6.12 UPnP/1.0 NETGEAR-UPNP/1.0
CONTENT-LENGTH:421

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body>
<s:Fault>
<faultcode>s:Client</faultcode>
<faultstring>UPnPError</faultstring>
<detail>
<UPnPError xmlns="urn:schemas-upnp-org:control-1-0">
<errorCode>402</errorCode>
<errorDescription>InvalidArgs</errorDescription></UPnPError>
</detail>
</s:Fault>
</s:Body>
</s:Envelope>
```

(b) An attempt to add a new port mapping entry. However, the server was unable to process the request.

Figure 5.14: Two distinct attempts at interacting with the SOAP part of the honeypot's UPnP service. Both received an erroneous reply. The attacking client is marked in red, while the honeypot's response is blue.

The first request is shown in fig. 5.14a. This attempts to perform the `GetGenericPortMappingEntry` action, which is used to retrieve configured port mappings from an IGD [Sta+10]. To get all mappings, control points must repetitively call this action with incremental indices until no more mappings are returned. Yet, the adversary immediately received an error. The reason for this is that they sent the POST request to the incorrect path. Thus, they attempted to start an information gathering routine on the honeypot but it short-circuited due to an unforeseen error. The `AddPortMapping` request in fig. 5.14b, on the other hand, was sent to the correct path. As explained in section 2.5.2, this attempts to add a new entry in the NAT table and can force the IGD to either expose additional internal ports, or become a proxy for the adversary to use. The example shown in fig. 5.14b attempts to map `172.217.20.78`'s port 80 to the honeypot's port 25473. In other words, they tried to proxy a remote web server through the honeypot. If this test would have succeeded, the attacker would have known that the service is vulnerable

and continued their attack. However, the UPnP service was unable to process the request and returned an application error mentioning invalid arguments. Debugging reveals that the Netgear® R7000's UPnP implementation does not allow for public IP addresses to be mapped. Indeed, this is the correct behaviour needed to halt the attempted attack. Note that in our implementation no additional ports can be accessed, even if the action would have succeeded, due to the firewall.

In conclusion, despite lacking variety in the types of attacks, UPnP is still actively being exploited. However, our choice of firmware is unfortunate as its service is not vulnerable. This prevented capturing further interactions. Experimentation shows that, if the adding of a port mapping would have succeeded, the request to fetch an added entry with `GetGenericPortMappingEntry` would have returned a mapping as expected. Emulating a UPnP service with high fidelity is thus feasible. Nevertheless, it requires some configuration to return values matching the honeypot's context due to not acting as the actual IGD of the network.

5.5 Lateral movement

As explained in section 4.1.1, our implementation features several low-interaction MQTT services. These are only accessible from within the honeypot private network. To end the analysis, we thus must check whether any intrusion, or subsequent infection, attempted to spread through the internal private network. To gain insight into the endpoints that communicated over the time period that the honeypots were online, all IP level conversations [SWL] are extracted from the network logs using Wireshark. A conversation is registered as soon as an endpoint sends at least one IP packet to another endpoint. Whether or not a response is sent and received, or the IP packet arrives at all, is irrelevant. This consequently defines entities by their IP addresses.

Table 5.6 lists all possible combinations of endpoint pairs along with the amount of data sent in each direction. This includes potential network scanning due to the above definition of IP level conversations. The Telnet proxy runs on the host and is thus listed separately from the IP camera, which is the emulated device that runs the environment to which Telnet provides access. Note that the data is extracted from the comparable time period only. Meaning, periods in which either of the two honeypot instances did not record network traffic are excluded. Yet, the data outside this time period does not contain any additional conversing source-target pairs. Following normal behaviour, only two pairs of internal endpoints are supposed to communicate. Those are the IP camera with the Telnet proxy, and the MQTT hub with the MQTT clients. And indeed, the data in the table shows only the expected conversations. As such, we conclude that the most prominently available IoT malware does not attempt to spread through private networks. Note that in section 5.2.3, several private IP addresses were found in malware samples. Given that other malware samples were able to run in the emulated environment, we reason that the IP addresses are not actually used during exploitation. Rather, they are meant as placeholders and replaced at runtime. Further research of the strings that the IP addresses are part of reveals them to be part of publicly available Proof of Concepts (PoC) to exploit related vulnerabilities. This confirms the hypothesis of the IPs being placeholders.

	External IPs	Telnet proxy	IP camera	Router	MQTT hub	MQTT clients
External IPs	/	0/0	0.34/0.28	0.42/0.36	0/0	0/0
Telnet proxy	0/0	/	1.80/3.09	0/0	0/0	0/0
IP camera	52.65/26.57	7.16/6.41	/	0/0	0/0	0/0
Router	0.62/0.81	0/0	0/0	/	0/0	0/0
MQTT hub	0/0	0/0	0/0	0/0	/	0.10/0.09
MQTT clients	0/0	0/0	0/0	0/0	0.15/0.15	/

Table 5.6: GigaBytes of data sent between entities over the comparable duration of the experiment. The source entities are listed in the rows, while the columns list the targets. Cells containing amounts of sent data list first the values for the AWS EC2 instance and then for the OVH VPS instance, separated by a /. A cell with only a / means that no applicable data exists.

Chapter 6

Conclusion and future work

In this thesis we implemented, deployed, and eventually analysed the data of a high interaction IoT honeypot that exposes consumer firmware. To achieve this, we first studied why IoT devices have notoriously bad security. This turns out to be a combination of three factors. Namely, limited hardware resources which hampers the implementation of sophisticated security features, lack of human knowledge due to the combination of various technologies and areas of expertise, and financial interests resulting in security being given less thought altogether. This led into learning about the workings of IoT malware, which can be divided into several stages. Furthermore, we studied how several popular IoT application layer protocols could be exploited. It became evident that a security by design approach, along with a restrictive design, is required to prevent both inherently vulnerable devices, as well as protocols. Then, to be able to design and implement the honeypot, an understanding of their general structure and core principles had to be acquired. Consideration and care must be given to the design and implementation of both the decoy, the part of the honeypot that adversaries end up interacting with, and the captor, which handles the data, in order to capture information relevant to the operator's goals. However, hosting consumer firmware turns out to be complicated due to devices being highly customised, which in turn results in heterogeneity and tight coupling between software and hardware. Last but not least, we created our own classification for firmware re-hosting techniques with a focus on their applicability in building a honeypot. The techniques were reviewed, and an evaluation of their practicality and applicability for building a honeypot was performed.

With the knowledge gained from the literature study, through implementing the honeypot, and from the data analysis, we are now ready to answer our research questions:

1. What techniques and/or malware do adversaries employ during attacks involving IoT devices?
   The different steps that IoT malware performs, along with examples, are discussed in section 2.4.2 based on prior work. Our captured data confirms their findings, and its analysis explains several examples in practice. The main takeaway from these discussions is that, despite IoT malware employing techniques similar to traditional malware targeting PCs or servers, the implementations tend to be less advanced. Yet, they have a big impact due to the security malpractices in IoT devices. As opposed to traditional malware, however, the amount of families of IoT malware are limited. They consequently all share similarities. Malware samples are updated frequently. This is done to add new features, and to change the IP addresses of the C&C and payload hosting servers. Nevertheless, the act of re-infecting victims to replace previous malware instances, along with the intrusion process being heavily automated, ensures a high infection rate. Lastly, the age of the used techniques is irrelevant as long as they are simple to automate and effective, as is demonstrated by the usage of old CVEs and UPnP/SSDP being abused.

2. Given that IoT devices are made to solve specific problems and thus differ wildly, to what extent do adversaries adapt to their targets. In other words, are the employed techniques and/or malware targeted?
   The differences between exploitation of distinct devices is relatively small. These include, but are not limited to, using device-specific credentials, CVEs, paths to files, commands to escape the vendors' restricted shell environments, and considering the victim's device architecture. Reducing the amount of properties that need to be considered between devices, while simultaneously covering

as many devices as possible for properties that can not be avoided, is of importance for malware creators. To this end, adversaries leverage the fact that Type 1 devices use generic, Unix-like OSes. It allows them to create a single malware binary that is as generic as possible. This increases the possibility of infecting a target, and consequently increases the rate at which the malware spreads.

3. Do adversaries attempt lateral movement through an IoT network, and how effective are they at it?
   The analysis of our captured data in section 5.5 reveals no attempts at lateral movement by adversaries. This is a surprising finding as connectedness, and thus having multiple devices in a network, is a selling point of the IoT paradigm. Given the generally weak security of devices, they should prove to be easy targets for adversaries. However, our honeypot instances only captured automated attacks. This conclusion thus only applies to malware binaries belonging to the captured families, i.e. Mirai and Gafgyt. A manual attack might be conducted differently.

4. Is state of the art firmware re-hosting usable to build a believable, high fidelity IoT honeypot?
   As seen in section 3.4.5, not every approach to firmware re-hosting can be used to build practical, high interaction honeypots. The primary challenge encountered by peripheral forwarding and virtual peripheral is lack of fidelity. However, the latter approach is the youngest across the board, which leaves room for improvement. Furthermore, the usable techniques, i.e. full device proxy and full system re-hosting, differ greatly. Their positives and negatives must be considered in the context of the honeypot operator's requirements. As for our implementation, built upon full system re-hosting, it was effective at capturing data covering the whole lifecycle of IoT malware. That is, from its initial access until it was used maliciously. Providing real services and system files increased the honeypot's fidelity. However, some difficulties were encountered, such as acquiring appropriate firmware and patching the emulated firmware in a manner that is believable. Furthermore, the used framework limited the implementation's fidelity as it required the usage of a least-privileged account, instead of providing intruders access to the `root` account. As it stands, if the requirement is to capture shell-based automated attacks with a honeypot that runs a RISC architecture, emulating OpenWrt is both less complex and has the same effectiveness. The reason being that current IoT malware performs barely any fingerprinting, and is designed to be as generic as possible. Yet, if the honeypot must look like a consumer device, i.e. contain appropriate files and branded services such as an administrative panel, full system re-hosting can be used, given a considerable time investment to set it up. This especially applies if the intent is to capture human interactions, as they are more sensitive to differences between a generic OpenWrt host and a consumer device. Nevertheless, full system re-hosting, and re-hosting in general, is still largely in its infancy.

Several complications were encountered during the implementation and analysis of our work. We use these to formulate important points to consider in future work. First, despite the decoy being the part of the honeypot that is seen by adversaries, the captor is similarly important and must receive ample care. Capturing data on multiple layers is important to fully reconstruct events that occurred on the honeypot. Also, every property of interest should be logged twice as to have a fallback in case of failing logging services. Furthermore, proper data control is paramount. Our intention is to study abuse in order to prevent it in the future. As such, utmost care must be taken to not inconvenience others. Related to data processing, when building a honeypot using existing firmware, the honeypot operator must be aware of the data generated by pre-existing services. For example, a baseline can be recorded in a controlled environment, and afterwards used to filter the noise from the actual maliciously generated honeypot data.

Next, hosting can impact the believability of the resulting honeypot instance. The provider and prior usage of the host contribute to this. Then, acquiring the firmware to use for the honeypot implementation is not simple. Aside from vendors not making blobs easily accessible, the current state of the art, full system re-hosting frameworks are not capable of emulating all firmware samples. Furthermore, the project at hand might require the firmware to include a specific vulnerability or (version of) software. Last but not least, full system re-hosting has inherent limitations due to no actual hardware being available. Patching these shortcomings is not to be taken lightly as they are the details that make or break whether humans continue interacting with the honeypot.

Throughout this work, we also encountered questions and difficulties that were either not answered with the data that we captured, or out of the scope of this thesis. We list these as research questions to be answered in future work. Note that the answers to our own research questions are based on the technology, both utilised frameworks and malware, available at the time. Thus, future work will also

have to re-visit these in due time.

- Do adversaries, when manually attacking a device, attempt lateral movement through an IoT network, and how effective are they at it?
- How can firmware re-hosting be improved, both in general and with a focus on building honeypots?
- How can firmware acquisition for security research purposes be improved? Is reverse engineering companion apps a viable option? And how effective is full system re-hosting at emulating the firmware acquired in this manner?

All in all, writing this thesis taught us a lot about the world of IoT security and cyber defence. However, it was difficult not to stray too far from the intended goals. Due to the interconnectedness of various disciplines, every resource would only increase the amount of ideas and previously unknown concepts, and consequently widen the scope of our research. As such, it was difficult to choose what should and should not be included in this thesis. While researching on our own taught us a lot, a drawback was that not considering something beforehand would come back later to bite us. One such example is not creating a baseline of the pre-existing services. Luckily, our mistakes were manageable in post. We hope that our lessons learned will benefit future work.

Bibliography

[Sin21] Satyajit Sinha. *State of IoT 2021: Number of connected IoT devices growing 9% to 12.3 billion globally, cellular IoT now surpassing 2 billion.* Accessed on: 2022-03-27. Sept. 2021. URL: https://iot-analytics.com/number-connected-iot-devices/.

[Gie19] Dennis Giese. *DEFCON 27 IoT Village - Dennis Giese - Privacy leaks in smart devices: Extracting data from used smart home devices.* Accessed on: 2022-03-27. Aug. 2019. URL: https://dontvacuum.me/talks/DEFCON27-IoT-Village/DEFCON27-IoT-Village_Dennis-Giese-Privacy-leaks-in-smart-devices.pdf.

[Ren+19] Jingjing Ren et al. 'Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach'. In: *Proceedings of the Internet Measurement Conference*. IMC '19 (2019), pp. 267–279. DOI: 10.1145/3355369.3355577.

[Cim17] Catalin Cimpanu. *A Hacker Just Pwned Over 150,000 Printers Left Exposed Online.* Accessed on: 2022-03-27. Feb. 2017. URL: https://www.bleepingcomputer.com/news/security/a-hacker-just-pwned-over-150-000-printers-left-exposed-online/.

[Kre16a] Brian Krebs. *KrebsOnSecurity Hit With Record DDoS.* Accessed on: 2022-03-27. Sept. 2016. URL: https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/.

[OVH16] OVH. *OVH News - The DDoS that didn't break the camel's VAC*.* Accessed on: 2022-03-27. Oct. 2016. URL: https://www.ovh.com/us/news/articles/a2367.the-ddos-that-didnt-break-the-camels-vac/, archived at https://web.archive.org/web/20170226233848/https://www.ovh.com/us/news/articles/a2367.the-ddos-that-didnt-break-the-camels-vac on 26th Feb. 2017.

[Int16] Flashpoint Intel. *An After-Action Analysis of the Mirai Botnet Attacks on Dyn.* Accessed on: 2022-03-27. Oct. 2016. URL: https://www.flashpoint-intel.com/cybercrime-forums-fraud/action-analysis-mirai-botnet-attacks-dyn/.

[GY21] Vivek Ganti and Omer Yoachimik. *A Brief History of the Meris Botnet.* Accessed on: 2022-03-27. Nov. 2021. URL: https://blog.cloudflare.com/meris-botnet/.

[Kay+20] Golam Kayas et al. 'An Overview of UPnP-based IoT Security: Threats, Vulnerabilities, and Prospective Solutions'. In: *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)* (2020), pp. 0452–0460. DOI: 10.1109/IEMCON51383.2020.9284885.

[Bou+20] Noureddine Boucif et al. 'Crushing the Wave - new Z-Wave vulnerabilities exposed'. In: *CoRR* abs/2001.08497 (2020). DOI: 10.48550/arXiv.2001.08497.

[San+21] Daniel dos Santos et al. *Amnesia: 33 How TCP/IP Stacks Breed Critical Vulnerabilities in IoT, OT and IT Devices.* Accessed on: 2022-02-27. 2021. URL: https://i.blackhat.com/eu-20/Wednesday/eu-20-dosSantos-How-Embedded-TCPIP-Stacks-Breed-Critical-Vulnerabilities-wp.pdf.

[Shu+19] Gao Shupeng et al. *All the 4G Modules Could be Hacked.* Accessed on: 2022-02-27. Aug. 2019. URL: https://i.blackhat.com/USA-19/Wednesday/us-19-Shupeng-All-The-4G-Modules-Could-Be-Hacked.pdf.

[Fan+18a] Wenjun Fan et al. 'Enabling an Anatomic View to Investigate Honeypot Systems: A Survey'. In: *IEEE Systems Journal* 12.4 (2018), pp. 3906–3919. DOI: 10.1109/JSYST.2017.2762161.

[Mai+11] Abhishek Mairh et al. 'Honeypot in Network Security: A Survey'. In: *Proceedings of the 2011 International Conference on Communication, Computing & Security.* ICCCS '11 (2011), pp. 600–605. DOI: 10.1145/1947940.1948065.

[Fra+21] Javier Franco et al. 'A Survey of Honeypots and Honeynets for Internet of Things, Industrial Internet of Things, and Cyber-Physical Systems'. In: *IEEE Communications Surveys Tutorials* (2021), pp. 1–1. DOI: 10.1109/COMST.2021.3106669.

[VC19] Alexander Vetterl and Richard Clayton. 'Honware: A Virtual Honeypot Framework for Capturing CPE and IoT Zero Days'. In: *2019 APWG Symposium on Electronic Crime Research (eCrime)* (2019), pp. 1–13. DOI: 10.1109/eCrime47957.2019.9037501.

[ES21] The European Union Agency for Network ENISA and Information Security. *Internet of things (IOT)*. Accessed on: 2021-10-18. Aug. 2021. URL: https://www.enisa.europa.eu/topics/iot-and-smart-infrastructures/iot/.

[IEE15] Internet Initiative IEEE. *Towards a definition of the Internet of Things (IoT)*. en. Accessed on: 2022-01-23. May 2015. URL: https://iot.ieee.org/images/files/pdf/IEEE_IoT_Towards_Definition_Internet_of_Things_Revision1_27MAY15.pdf.

[Gre+19] Christopher Greer et al. 'Cyber-Physical Systems and Internet of Things'. en. In: *NIST Special Publication* 1900.202 (Mar. 2019). DOI: 10.6028/NIST.SP.1900-202.

[Mue+18a] Marius Muench et al. 'What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices'. In: *Network and Distributed System Security (NDSS) Symposium 2018* (Jan. 2018). DOI: 10.14722/ndss.2018.23176.

[Wri+21] Christopher Wright et al. 'Challenges in Firmware Re-Hosting, Emulation, and Analysis'. In: *ACM Comput. Surv.* 54.1 (Jan. 2021). ISSN: 0360-0300. DOI: 10.1145/3423167.

[Wan17] KC Wang. *Embedded and Real-Time Operating Systems*. Springer, 2017, pp. 401–475. ISBN: 9783319515168.

[Lal13] Sanjay Lal. *Real World Multicore Embedded Systems*. Ed. by Bryon Moyer. Oxford: Newnes, 2013. Chap. 15, pp. 517–560. ISBN: 9780124160187. DOI: 10.1016/B978-0-12-416018-7.00015-8.

[Ant+17] Manos Antonakakis et al. 'Understanding the Mirai Botnet'. In: *26th USENIX Security Symposium (USENIX Security 17)* (Aug. 2017), pp. 1093–1110. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[MS19] Daniel Miessler and Craig Smith. *IoT Attack Surface Areas Project*. Accessed on: 2022-01-17. Nov. 2019. URL: https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Attack_Surface_Areas.

[Zha+14] Zhi-Kai Zhang et al. 'IoT Security: Ongoing Challenges and Research Opportunities'. In: *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications* (2014), pp. 230–234. DOI: 10.1109/SOCA.2014.58.

[ES17] The European Union Agency for Network ENISA and Information Security. *Baseline Security Recommendations for IoT*. en. Accessed on: 2021-10-21. Nov. 2017. URL: https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot/@@download/fullReport.

[Nes+19] Nataliia Neshenko et al. 'Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations'. In: *IEEE Communications Surveys Tutorials* 21.3 (2019), pp. 2702–2733. DOI: 10.1109/COMST.2019.2910750.

[Sug+20] Takeshi Sugawara et al. 'Light Commands: Laser-Based Audio Injection Attacks on Voice-Controllable Systems'. In: *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), pp. 2631–2648. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/sugawara.

[Pal20] Palo Alto Networks, Inc. *Impacts of Cyberattacks on IoT Devices*. Accessed on: 2022-03-31. 2020. URL: https://www.paloaltonetworks.com/content/dam/pan/en_US/assets/pdf/reports/impacts-of-cyberattacks-on-iot-devices.pdf.

[Sin+17] Saurabh Singh et al. 'Advanced lightweight encryption algorithms for IoT devices: survey, challenges and solutions'. In: *Journal of Ambient Intelligence and Humanized Computing* (May 2017), pp. 1–18. DOI: 10.1007/s12652-017-0494-4.

[Men+19] Francesca Meneghello et al. 'IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices'. In: *IEEE Internet of Things Journal* 6.5 (2019), pp. 8182–8201. DOI: 10.1109/JIOT.2019.2935189.

[Dan+19] Fan Dang et al. 'Understanding Fileless Attacks on Linux-Based IoT Devices with HoneyCloud'. In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '19 (2019), pp. 482–493. DOI: 10.1145/3307334.3326083.

[Alr+21] Omar Alrawi et al. 'The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle'. In: *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), pp. 3505–3522.

URL: https://www.usenix.org/conference/usenixsecurity21/presentation/alrawi-circle.

[LXC12] Jing Liu, Yang Xiao and C.L. Philip Chen. 'Authentication and Access Control in the Internet of Things'. In: *2012 32nd International Conference on Distributed Computing Systems Workshops* (2012), pp. 588–592. DOI: 10.1109/ICDCSW.2012.23.

[ano12] anonymous. *Internet Census 2012 - Port scanning /0 using insecure embedded devices - Carna Botnet*. Accessed on: 2022-03-19. 2012. URL: https://census2012.sourceforge.net/paper.html, archived at https://web.archive.org/web/20220319071815/http://census2012.sourceforge.net/paper.html on 19th Mar. 2022.

[Oua+17] Aafaf Ouaddah et al. 'Access control in the Internet of Things: Big challenges and new opportunities'. In: *Computer Networks* 112 (2017), pp. 237–262. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2016.11.007.

[MB18] Philipp Morgner and Zinaida Benenson. 'Exploring Security Economics in IoT Standardization Efforts'. In: *CoRR* abs/1810.12035 (2018). DOI: 10.48550/arXiv.1810.12035.

[Eur17] Secretariat-General European Commission. *JOINT COMMUNICATION TO THE EUROPEAN PARLIAMENT AND THE COUNCIL Resilience, Deterrence and Defence: Building strong cybersecurity for the EU*. Document 52017JC0450. Accessed on: 2022-03-29. Sept. 2017. URL: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:52017JC0450.

[Eur19] Council of the European Union European Parliament. *Regulation (EU) 2019/881 of the European Parliament and of the Council of 17 April 2019 on ENISA (the European Union Agency for Cybersecurity) and on information and communications technology cybersecurity certification and repealing Regulation (EU) No 526/2013 (Cybersecurity Act) (Text with EEA relevance)*. Document 32019R0881. Accessed on: 2022-03-29. Apr. 2019. URL: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32019R0881.

[Man17] Steve Mansfield-Devine. 'Fileless attacks: compromising targets without malware'. In: *Network Security* 2017.4 (2017), pp. 7–11. ISSN: 1353-4858. DOI: 10.1016/S1353-4858(17)30037-5.

[Kum+20] Sushil Kumar et al. 'An emerging threat Fileless malware: a survey and research challenges'. In: *Cybersecurity* 3.1 (2020), pp. 1–12. DOI: 10.1186/s42400-019-0043-x.

[Dan+21] Dansimp et al. *Fileless threats - Windows security — Microsoft Docs*. Accessed on: 2022-03-17. Oct. 2021. URL: https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/fileless-threats.

[San17] John Sanchez. *KOVTER: An Evolving Malware Gone Fileless - Security News*. Accessed on: 2022-03-17. Aug. 2017. URL: https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/kovter-an-evolving-malware-gone-fileless.

[MIT21] MITRE. *MITRE ATT&CK*. Accessed on: 2022-04-10. 2021. URL: https://attack.mitre.org/.

[Pa+15] Yin Minn Pa Pa et al. 'IoTPOT: Analysing the Rise of IoT Compromises'. In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (Aug. 2015). URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/pa.

[TOS21] Armin Ziaie Tabari, Xinming Ou and Anoop Singhal. 'What are Attackers after on IoT Devices? An approach based on a multi-phased multi-faceted IoT honeypot ecosystem and data clustering'. In: *CoRR* abs/2112.10974 (2021). DOI: 10.48550/arXiv.2112.10974.

[VKH19] Benjamin Vignau, Raphael Khoury and Sylvain Hallé. '10 Years of IoT Malware: A Feature-Based Taxonomy'. In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)* (2019), pp. 458–465. DOI: 10.1109/QRS-C.2019.00088.

[MS18] Lionel Metongnon and Ramin Sadre. 'Beyond Telnet: Prevalence of IoT Protocols in Telescope and Honeypot Measurements'. In: *Proceedings of the 2018 Workshop on Traffic Measurements for Cybersecurity*. WTMC '18 (2018), pp. 21–26. DOI: 10.1145/3229598.3229604.

[EP16] Sam Edwards and Ioannis Profetis. *Hajime: Analysis of a decentralized internet worm for IoT devices*. Accessed on: 2022-04-15. 2016. URL: http://security.rapiditynetworks.com/publications/2016-10-16/hajime.pdf, archived at https://web.archive.org/web/20210617095628/http://security.rapiditynetworks.com/publications/2016-10-16/hajime.pdf on 17th June 2021.

[Tor+21] Sadegh Torabi et al. 'A Strings-Based Similarity Analysis Approach for Characterizing IoT Malware and Inferring Their Underlying Relationships'. In: *IEEE Networking Letters* 3.3 (2021), pp. 161–165. DOI: 10.1109/LNET.2021.3076600.

[Mic19] Trend Micro. *Silex Malware Bricks IoT Devices with Weak Passwords*. Accessed on: 2022-04-01. June 2019. URL: https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/-silex-malware-bricks-iot-devices-with-weak-passwords.

[Mer18] Fernando Merces. *Miner Malware Targets IoT, Offered in the Underground*. Accessed on: 2022-04-01. May 2018. URL: https://www.trendmicro.com/en_us/research/18/e/cryptocurrency-mining-malware-targeting-iot-being-offered-in-the-underground.html.

[Ron+17] Eyal Ronen et al. 'IoT Goes Nuclear: Creating a ZigBee Chain Reaction'. In: *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 195–212. DOI: 10.1109/SP.2017.14.

[Jia+20] Yan Jia et al. 'Burglars' IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds'. In: *2020 IEEE Symposium on Security and Privacy (SP)* (2020), pp. 465–481. DOI: 10.1109/SP40000.2020.00051.

[MVQ18] Federico Maggi, Rainer Vosseler and Davide Quarta. *The Fragility of Industrial IoT's Data Backbone. Security and Privacy Issues in MQTT and CoAP Protocols*. Accessed on: 2022-03-01. 2018. URL: https://documents.trendmicro.com/assets/white_papers/wp-the-fragility-of-industrial-IoTs-data-backbone.pdf.

[SHB14] Zach Shelby, Klaus Hartke and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252. June 2014. DOI: 10.17487/RFC7252.

[Ban+19] Andrew Banks et al. *MQTT Version 5.0*. OASIS Standard. Accessed on: 2022-03-01. Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html.

[Bra97] Scott O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Mar. 1997. DOI: 10.17487/RFC2119.

[Wan+21] Qinying Wang et al. 'MPInspector: A Systematic and Automatic Approach for Evaluating the Security of IoT Messaging Protocols'. In: *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), pp. 4205–4222. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying.

[Fou22] The Open Connectivity Foundation. *OCF - UPnP Standards & Architecture*. Accessed on: 2022-03-01. 2022. URL: https://openconnectivity.org/developer/specifications/upnp-resources/upnp/.

[Cor12] IBM Corporation. *What is zero configuration networking? - IBM Documentation*. Accessed on: 2022-03-01. 2012. URL: https://www.ibm.com/docs/en/snips/4.6.0?topic=networking-what-is-zero-configuration.

[Don+15] Andrew Donoho et al. *UPnP Device Architecture 2.0*. Accessed on: 2022-03-01. 2015. URL: https://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf.

[Maj17a] Marek Majkowski. *Stupidly Simple DDoS Protocol (SSDP) generates 100 Gbps DDoS*. Accessed on: 2022-03-03. June 2017. URL: https://blog.cloudflare.com/ssdp-100gbps/.

[Squ08] Jonathan Squire. *Universal Plug and Play IGD - A Fox in the Hen House*. Accessed on: 2022-02-27. Aug. 2008. URL: https://www.blackhat.com/presentations/bh-usa-08/Squire/BH_US_08_Squire_A_Fox_in_the_Hen_House%20White%20Paper.pdf.

[Moo13] HD Moore. *Security Flaws in Universal Plug and Play: Unplug, Don't Play*. Accessed on: 2022-03-03. Jan. 2013. URL: https://information.rapid7.com/rs/411-NAK-970/images/SecurityFlawsUPnP%20%281%29.pdf.

[Res18] Akamai Threat Research. *UPnProxy: Blackhat Proxies via NAT Injections*. Accessed on: 2022-04-05. Mar. 2018. URL: https://www.akamai.com/site/en/documents/research-paper/upnproxy-blackhat-proxies-via-nat-injections-white-paper.pdf.

[BPT15] Mike Belshe, Roberto Peon and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540.

[FK11] Sheila Frankel and Suresh Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071. Feb. 2011. DOI: 10.17487/RFC6071.

[RM12] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. Jan. 2012. DOI: 10.17487/RFC6347.

[Har+17] Asma Haroon et al. 'E-Lithe: A Lightweight Secure DTLS for IoT'. In: *2017 IEEE 86th Vehicular Technology Conference (VTC-Fall)* (2017), pp. 1–5. DOI: 10.1109/VTCFall.2017.8288362.

[MEB16] Yassine Maleh, Abdellah Ezzati and Mustapha Belaissaoui. 'An enhanced DTLS protocol for Internet of Things applications'. In: *2016 International Conference on Wireless Networks*

and Mobile Communications (WINCOM) (2016), pp. 168–173. DOI: 10.1109/WINCOM.2016.7777209.

[Cap+15] Angelo Capossele et al. 'Security as a CoAP resource: An optimized DTLS implementation for the IoT'. In: *2015 IEEE International Conference on Communications (ICC)* (2015), pp. 549–554. DOI: 10.1109/ICC.2015.7248379.

[Par20] Chang-Seop Park. 'Security Architecture for Secure Multicast CoAP Applications'. In: *IEEE Internet of Things Journal* 7.4 (2020), pp. 3441–3452. DOI: 10.1109/JIOT.2020.2970175.

[Spi02] Lance Spitzner. *Honeypots: tracking hackers*. Vol. 1. Addison-Wesley, 2002. ISBN: 9780321108951.

[ES12] The European Union Agency for Network ENISA and Information Security. *Proactive detection of security incidents II - Honeypots*. en. Accessed on: 2021-10-21. Nov. 2012. URL: https://www.enisa.europa.eu/publications/proactive-detection-of-security-incidents-II-honeypots/@@download/fullReport.

[Fan+18b] Wenjun Fan et al. 'Enabling an Anatomic View to Investigate Honeypot Systems: A Survey'. In: *IEEE Systems Journal* 12.4 (2018), pp. 3906–3919. DOI: 10.1109/JSYST.2017.2762161.

[Spi03] Lance Spitzner. *Honeytokens: The Other Honeypot*. Accessed on: 2022-01-23. July 2003. URL: https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=74450cf5-2f11-48c5-8d92-4687f5978988&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments.

[Ber+11] Maya Bercovitch et al. 'HoneyGen: An automated honeytokens generator'. In: *Proceedings of 2011 IEEE International Conference on Intelligence and Security Informatics* (2011), pp. 131–136. DOI: 10.1109/ISI.2011.5984063.

[SWK+07] Christian Seifert, Ian Welch, Peter Komisarczuk et al. *Honeyc-the low-interaction client honeypot*. Accessed on: 2022-01-23. 2007. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.61.6882&rep=rep1&type=pdf.

[Naw+16] Marcin Nawrocki et al. 'A Survey on Honeypot Software and Data Analysis'. In: *CoRR* abs/1608.06249 (2016). DOI: 10.48550/arXiv.1608.06249.

[BN17] Timothy Barron and Nick Nikiforakis. 'Picky Attackers: Quantifying the Role of System Properties on Intruder Behavior'. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC 2017 (2017), pp. 387–398. DOI: 10.1145/3134600.3134614.

[Mis+18] Chedy Missaoui et al. 'Who is reusing stolen passwords? An empirical study on stolen passwords and countermeasures'. In: *International Symposium on Cyberspace Safety and Security* (2018), pp. 3–17. DOI: 10.1007/978-3-030-01689-0_1.

[Fra+18a] Daniel Fraunholz et al. 'Hack My Company: An Empirical Assessment of Post-Exploitation Behavior and Lateral Movement in Cloud Environments'. In: CECC 2018 (2018). DOI: 10.1145/3277570.3277573.

[LOS16] Martin Lazarov, Jeremiah Onaolapo and Gianluca Stringhini. 'Honey Sheets: What Happens to Leaked Google Spreadsheets?' In: *9th Workshop on Cyber Security Experimentation and Test (CSET 16)* (Aug. 2016). URL: https://www.usenix.org/conference/cset16/workshop-program/presentation/lazarov.

[Hil+20] Stephen Hilt et al. *Caught in the Act: Running a Realistic Factory Honeypot to Capture Real Threats*. Accessed on: 2022-03-14. 2020. URL: https://documents.trendmicro.com/assets/white_papers/wp-caught-in-the-act-running-a-realistic-factory-honeypot-to-capture-real-threats.pdf.

[OMS16] Jeremiah Onaolapo, Enrico Mariconti and Gianluca Stringhini. 'What Happens After You Are Pwnd: Understanding the Use of Leaked Webmail Credentials in the Wild'. In: IMC '16 (2016), pp. 65–79. DOI: 10.1145/2987443.2987475.

[Aki+18] Mitsuaki Akiyama et al. 'HoneyCirculator: distributing credential honeytoken for introspection of web-based attack cycle'. In: *International Journal of Information Security* 17 (Apr. 2018). DOI: 10.1007/s10207-017-0361-5.

[Mor+19] Shun Morishita et al. 'Detect Me If You... Oh Wait. An Internet-Wide View of Self-Revealing Honeypots'. In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)* (2019), pp. 134–143. URL: http://yoshioka.ynu.ac.jp/papers/IM2019-honeypot.pdf.

[VC18] Alexander Vetterl and Richard Clayton. 'Bitter Harvest: Systematically Fingerprinting Low- and Medium-interaction Honeypots at Internet Scale'. In: *12th USENIX Workshop on Offensive Technologies (WOOT 18)* (Aug. 2018). URL: https://www.usenix.org/conference/woot18/presentation/vetterl.

[mor17] morgajp. *Issue: Missing response to "cat /bin/echo" command*. Accessed on: 2022-01-28. Jan. 2017. URL: https://github.com/Cymmetria/MTPot/issues/9.

[SM15] Valerio Selis and A. Marshall. 'MEDA: a Machine Emulation Detection Algorithm'. In: *Proceedings of the 12th International Conference on Security and Cryptography* (July 2015). DOI: 10.5220/0005535202280235.

[HR05] T. Holz and F. Raynal. 'Detecting honeypots and other suspicious environments'. In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop* (2005), pp. 29–36. DOI: 10.1109/IAW.2005.1495930.

[Jan+19] Daehee Jang et al. 'Rethinking anti-emulation techniques for large-scale software deployment'. In: *Computers & Security* 83 (2019), pp. 182–200. ISSN: 0167-4048. DOI: 10.1016/j.cose.2019.02.005.

[Gar+07] Tal Garfinkel et al. 'Compatibility is Not Transparency: VMM Detection Myths and Realities'. In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*. HOTOS'07 (2007). URL: https://www.cs.cmu.edu/~jfrankli/hotos07/vmm_detection_hotos07.pdf.

[Fer07] Peter Ferrie. *Attacks on Virtual Machine Emulators*. Accessed on: 2022-03-01. 2007. URL: https://pferrie.tripod.com/papers/attacks2.pdf.

[Ked+17] Alexander Kedrowitsch et al. 'A First Look: Using Linux Containers for Deceptive Honeypots'. In: *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*. SafeConfig '17 (2017), pp. 15–22. DOI: 10.1145/3140368.3140371.

[Mir+17] Najmeh Miramirkhani et al. 'Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts'. In: *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 1009–1024. DOI: 10.1109/SP.2017.42.

[CNZ20] Humberto Carvalho, Geoffrey Nelissen and Pavel Zaykov. 'mcQEMU: Time-Accurate Simulation of Multi-core platforms using QEMU'. In: *2020 23rd Euromicro Conference on Digital System Design (DSD)* (2020), pp. 81–88. DOI: 10.1109/DSD51259.2020.00024.

[Pro04] The Honeynet Project. *Charter - Honeynet Definitions, Requirements, and Standards*. Accessed on: 2021-10-24. Oct. 2004. URL: https://honeynet.onofri.org/alliance/requirements.html.

[Pro06] The Honeynet Project. *Know Your Enemy: Honeynets*. Accessed on: 2022-01-23. May 2006. URL: https://project.honeynet.org/papers/honeynet/index.html, archived at https://web.archive.org/web/20120905181856/https://project.honeynet.org/papers/honeynet/index.html on 5th Sept. 2012.

[Vas+15] Emmanouil Vasilomanolakis et al. 'A Honeypot-Driven Cyber Incident Monitor: Lessons Learned and Steps Ahead'. In: *Proceedings of the 8th International Conference on Security of Information and Networks*. SIN '15 (2015), pp. 158–164. DOI: 10.1145/2799979.2799999.

[Fas+21] Andrew Fasano et al. 'SoK: Enabling Security Analyses of Embedded Systems via Rehosting'. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (2021), pp. 687–701. DOI: 10.1145/3433210.3453093.

[Cab+19] Warren Cabral et al. 'Review and Analysis of Cowrie Artefacts and Their Potential to be Used Deceptively'. In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)* (2019), pp. 166–171. DOI: 10.1109/CSCI49370.2019.00035.

[Fra+18b] Daniel Fraunholz et al. 'Introducing Falcom: A Multifunctional High-Interaction Honeypot Framework for Industrial and Embedded Applications'. In: *2018 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)* (2018), pp. 1–8. DOI: 10.1109/CyberSecPODS.2018.8560675.

[Hak+18] Muhammad A. Hakim et al. 'U-PoT: A Honeypot Framework for UPnP-Based IoT Devices'. In: *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)* (2018), pp. 1–8. DOI: 10.1109/PCCC.2018.8711321.

[WSK18] Meng Wang, Javier Santillan and Fernando Kuipers. 'ThingPot: an interactive Internet-of-Things honeypot'. In: *CoRR* abs/1807.04114 (2018). DOI: 10.48550/arXiv.1807.04114.

[Spe+21] Chad Spensky et al. 'Conware: Automated Modeling of Hardware Peripherals'. In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (2021), pp. 95–109. DOI: 10.1145/3433210.3437532.

[Ern03] Michael D. Ernst. *Static and dynamic analysis: Synergy and duality*. Accessed on: 2022-02-07. 2003. URL: https://www.cs.nmsu.edu/~jcook/woda2003/papers/Ernst.pdf.

[Zho19] Ye Zhou. 'Chameleon: Towards Adaptive Honeypot for Internet of Things'. In: *Proceedings of the ACM Turing Celebration Conference - China*. ACM TURC '19 (2019). DOI: 10.1145/3321408.3321589.

[Luo+17] Tongbo Luo et al. *Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices*. Accessed on: 2022-02-27. 2017. URL: https://www.blackhat.com/docs/us-17/thursday/us-17-Luo-Iotcandyjar-Towards-An-Intelligent-Interaction-Honeypot-For-IoT-Devices-wp.pdf.

[Gua+17] Juan David Guarnizo et al. 'SIPHON: Towards Scalable High-Interaction Physical Honeypots'. In: *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*. CPSS '17 (2017), pp. 57–68. DOI: 10.1145/3055186.3055192.

[SGA07] Michael Sutton, Adam Greene and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007. ISBN: 9780321680853.

[Zad+14] Jonas Zaddach et al. 'Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares'. In: (Feb. 2014). DOI: 10.14722/ndss.2014.23229.

[KPK14] Markus Kammerstetter, Christian Platzer and Wolfgang Kastner. 'Prospect: Peripheral Proxying Supported Embedded Code Testing'. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '14 (2014), pp. 329–340. DOI: 10.1145/2590296.2590301.

[K J96] Michael K. Johnson. *Device Driver Basics*. Accessed on: 2022-02-07. 1996. URL: https://tldp.org/LDP/khg/HyperNews/get/devices/basics.html.

[Mue+18b] Marius Muench et al. 'Avatar 2 : A Multi-Target Orchestration Platform'. In: *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* 18 (2018), pp. 1–11. DOI: 10.14722/bar.2018.23017.

[Gus+19] Eric Gustafson et al. 'Toward the Analysis of Embedded Firmware through Automated Rehosting'. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)* (Sept. 2019), pp. 135–150. URL: https://www.usenix.org/conference/raid2019/presentation/gustafson.

[Joh+21] Evan Johnson et al. 'Jetset: Targeted Firmware Rehosting for Embedded Systems'. In: *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), pp. 321–338. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson.

[Zho+21] Wei Zhou et al. 'Automatic Firmware Emulation through Invalidity-guided Knowledge Inference'. In: *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), pp. 2007–2024. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou.

[Kin76] James C. King. 'Symbolic Execution and Program Testing'. In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[Kim+20] Mingeun Kim et al. 'FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis'. In: *Annual Computer Security Applications Conference*. ACSAC '20 (2020), pp. 733–745. DOI: 10.1145/3427228.3427294.

[CZF15] Andrei Costin, Apostolis Zarras and Aurélien Francillon. 'Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces'. In: *CoRR* abs/1511.03609 (2015). DOI: 10.48550/arXiv.1511.03609.

[Che+16] Daming Chen et al. 'Towards Automated Dynamic Analysis for Linux-based Embedded Firmware'. In: *Network and Distributed System Security (NDSS) Symposium 2016* (Jan. 2016). DOI: 10.14722/ndss.2016.23415.

[iot21] Eclipse iot. *IoT & Edge: Developer Survey Report*. Accessed on: 2022-03-19. Dec. 2021. URL: https://f.hubspotusercontent10.net/hubfs/5413615/IoT%20&%20Edge%20Developer%20Survey%20Report%20-%202021.pdf.

[Bow+09] Terrehon Bowden et al. *The /proc Filesystem - Chapter 4: Configuring procfs*. Accessed on: 2022-03-22. June 2009. URL: https://www.kernel.org/doc/html/latest/filesystems/proc.html#chapter-4-configuring-procfs.

[Maj17b] Marek Majkowski. *Reflections on reflection (attacks)*. Accessed on: 2022-03-19. May 2017. URL: https://blog.cloudflare.com/reflections-on-reflections/.

[Ham+16] Ryan Hamilton et al. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft. Work in Progress. Oct. 2016. URL: https://datatracker.ietf.org/doc/html/draft-hamilton-quic-transport-protocol-01.

[Kre16b] Brian Krebs. *Source Code for IoT Botnet 'Mirai' Released*. Accessed on: 2022-04-28. Oct. 2016. URL: https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/.

[Lev16] Level 3 Threat Research Labs. *Attack of Things! - Beyond Bandwidth*. Accessed on: 2022-05-03. Sept. 2016. URL: https://blog.level3.com/security/attack-of-things/, archived at https://web.archive.org/web/20170126131627/http://blog.level3.com/security/attack-of-things/ on 26th Jan. 2017.

[Dav19] Asher Davila. *Home & Small Office Wireless Routers Exploited to Attack Gaming Servers*. Accessed on: 2022-05-03. Oct. 2019. URL: https://unit42.paloaltonetworks.com/home-small-office-wireless-routers-exploited-to-attack-gaming-servers/.

[Sea21] Tara Seals. *Gafgyt Botnet Lifts DDoS Tricks from Mirai — Threatpost*. Accessed on: 2022-05-03. Apr. 2021. URL: https://threatpost.com/gafgyt-botnet-ddos-mirai/165424/.

[Ull14] Johannes Ullrich. *InfoSec Handlers Diary Blog - SANS Internet Storm Center*. Accessed on: 2022-05-03. Feb. 2014. URL: https://isc.sans.edu/diary/Linksys+Worm+%22TheMoon%22+Summary%3A+What+we+know+so+far/17633.

[Kim20] Paul Kimayong. *IoT botnet exploiting TVT Shenzhen DVRs still lingers — Official Juniper Networks Blogs*. Accessed on: 2022-05-03. Mar. 2020. URL: https://blogs.juniper.net/en-us/threat-research/iot-botnet-exploiting-tvt-shenzhen-dvrs-still-lingers.

[WO07] Christer Weingel and Jakob Oestergaard. *The Linux Watchdog driver API*. Accessed on: 2022-05-05. May 2007. URL: https://www.kernel.org/doc/Documentation/watchdog/watchdog-api.txt.

[Voo+19] Doron Voolf et al. *Gafgyt Targeting Huawei and Asus Routers and Killing Off Rival IoT Botnets*. Accessed on: 2022-05-03. Dec. 2019. URL: https://www.f5.com/labs/articles/threat-intelligence/gafgyt-targeting-huawei-and-asus-routers-and-killing-off-rival-iot-botnets/.

[Sta+10] Barbara Stark et al. *WANIPConnection:2 Service - For UPnP Version 1.0*. Accessed on: 2022-05-09. 2010. URL: http://upnp.org/specs/gw/UPnP-gw-WANIPConnection-v2-Service.pdf.

[SWL] Richard Sharpe, Ed Warnicke and Ulf Lamping. *8.5. Conversations*. Accessed on: 2022-05-10. URL: https://www.wireshark.org/docs/wsug_html_chunked/ChStatConversations.html.

Appendix A

Nederlandse samenvatting

A.1 Introductie

Het aantal apparaten die deel uitmaken van het internet der dingen, ook wel Internet of Things (IoT) genoemd, is in de recente jaren sterk toegenomen. Het doel van deze apparaten is om onze levenskwaliteit te verbeteren. Dit wordt mogelijk gemaakt door het sterk integreren van computers met dagdagelijkse apparatuur. Te sterk afhankelijk worden van deze apparaten is echter gevaarlijk. Indien ze worden gehackt kan dit zowel onze privacy als fysieke veiligheid ondermijnen.

Om te kunnen verdedigen tegen het misbruik van deze apparaten moet er eerst geweten worden hoe hackers te werk gaan. Het achterhalen van de grondoorzaken helpt verder niet enkel met het verstaan van het geobserveerd misbruik, maar ook bij het formuleren van preventieve maatregelen. Informatie over misbruik kan vergaard worden door het analyseren van log data. Data van productiesystemen opvragen bij gehackte bedrijven of individuen is echter niet doenbaar. Deze kunnen namelijk persoonlijke gegevens bevatten, onvolledig zijn, of hun integriteit verloren hebben door de aanval. Een gecontroleerde omgeving is dus benodigd om informatie te verzamelen. Zo een omgeving noemt een honeypot.

Honeypots met focus op het nabootsen van IoT-apparaten zijn geen nieuw concept. Eerdere projecten zijn echter gebrekkig. Zo focussen ze zich op individuele services en/of apparaten, voorzien geen echte shell-omgeving, of doen zich niet voor als consumentenapparatuur. Met andere woorden, ze zijn niet waarheidsgetrouw. Het doel van deze thesis is dan ook het analyseren van het IoT-bedreigingslandschap met behulp van waarheidsgetrouwe honeypots op basis van consumentenfirmware. Het blijkt echter dat het virtualiseren van consumentenfirmware niet eenvoudig is. Onze onderzoeksvragen zijn dus als volgt:

- Welke technieken en/of malware gebruiken hackers bij aanvallen waarbij IoT-apparaten betrokken zijn?
- Slimme apparaten worden gemaakt om zeer specifieke problemen op te lossen. Ze verschillen dus veel van elkaar. Hoe passen hackers zich aan aan deze verschillen?
- Proberen hackers zich door privénetwerken te verspreiden, en hoe goed lukt hen dat?
- Zijn state of the art virtualisatietechnieken bruikbaar om geloofwaardige honeypots te bouwen op basis van consumentenfirmware?

A.2 Internet der dingen (IoT)

Ondanks dat elk apparaat gemaakt is voor een specifiek doel, en ze dus sterk van elkaar verschillen, kunnen er toch structurele gelijkenissen geïdentificeerd worden. Op een technisch niveau kan er een verschil worden gemaakt in de manier waarop de firmware van een apparaat functioneert. Hierin bestaan er drie types. Type 1 maakt gebruik van een generiek besturingssysteem dat is aangepast geworden om performant op zwakkere hardware te draaien. Type 2 systemen maken gebruik van speciale besturingssystemen. Deze zijn ontworpen met specifieke use cases in gedachten. Ten laatste, Type 3 apparaten gebruiken geen besturingssysteem. Het beheren van het systeem ligt compleet in de handen van de

ontwikkelaar. Deze thesis focust zich verder enkel op Type 1 apparaten. Deze zijn op het moment van schrijven populair doordat de flexibiliteit van het besturingssysteem meer complexiteit in het eindproduct toelaat.

Eerdere studies constateerden dat IoT-apparaten zeer onveilig zijn. De reden hiervoor is dat de best practices van de laatste twee decennia niet toereikend worden toegepast. Volgende achterliggende redenen werden ontdekt:

- De hardware beschikt over minder mogelijkheden vergeleken met traditionele hardware, zoals servers en pc's. Het gebruiken van een batterij voor stroomtoevoer is hier een voorbeeld van. Deze kan snel leeg getrokken worden door excessief gebruik van de processor. Zo is bijvoorbeeld het implementeren van sterke encryptie niet mogelijk. Dit beïnvloedt dan weer data opslag en het veilig versturen hiervan, wat detectie van ongewone activiteiten compliceert.
- Menselijke kennis is niet altijd adequaat. IoT combineert meerdere technologieën, en dus expertisegebieden. Elk project is anders doordat er geen standaardsysteemarchitectuur en weinig abstractielagen bestaan. Het kan dus niet verwacht worden van ontwikkelaars dat ze van alles op de hoogte zijn. Verder zijn veel eindgebruikers niet technisch ingesteld en hebben ze weinig kennis van beveiligingspraktijken. Als gevolg worden apparaten niet geüpdatet, standaardwachtwoorden niet aangepast, enzoverder.
- Er ontbreekt ook een drijfveer voor bedrijven om in veiligheid te investeren. Consumenten blijken meer geïnteresseerd te zijn in bruikbaarheid, functies en interoperabiliteit dan in veiligheid. Snel iets op de markt brengen is financieel voordeliger dan extra tijd en geld te investeren in het testen van een nieuw product.

Cyberaanvallen worden in meerdere stappen uitgevoerd. Elke stap kan op verschillende manieren worden gerealiseerd. Eerdere studies van malware die zich richtten op IoT onthulden dat de gebruikte technieken overgenomen zijn van traditionele malware. Het verschil is dat de implementaties minder gesofisticeerd zijn, vergeleken met de traditionele versies. In combinatie met het gebrek aangepaste beveiligingsmaatregelen zijn de gevolgen echter zwaarder. De stappen genomen door malware die zich richt op IoT zijn als volgt:

1. Initiële toegang: Netwerken worden gescand voor mogelijke onveilige services. De scanner wordt verwerkt in de malware die op het slachtoffer wordt geïnstalleerd. Dit resulteert in een exponentiële groei van het aantal IP-adressen die gescand kunnen worden binnen een bepaalde tijd, per nieuw geïnfecteerd apparaat.
2. Uitvoeren van instructies: Doordat er weinig menselijke interactie is met IoT-apparaten, wordt er sterk ingezet op het automatiseren van aanvallen. Binnendringen in apparaten gebeurt voornamelijk dan ook met behulp van standaardinloggegevens en publiek bekende kwetsbaarheden.
3. Persistentie: Na het infecteren worden enkele acties ondernomen om te verzekeren dat het apparaat ook geïnfecteerd blijft. Bijvoorbeeld, het herstarten zou ervoor zorgen dat het malwareproces gestopt wordt. Door het gebrek aan menselijk toezicht moeten IoT-apparaten zichzelf controleren en herstarten indien iets verkeerd gaat. Malware kan deze controle beïnvloeden om zo het herstarten te voorkomen.
4. Detectie ontwijken: Malware probeert zo veel mogelijk onder de radar te blijven, alsook zijn sporen te wissen. Dit verkleint de kans dat de malwarecampagne wordt opgedoekt. Het hernoemen van het malwareproces en het toepassen van obfuscatie op de binaire code zijn enkele technieken om detectie te ontwijken.
5. Informatie vergaren: Een geïnfecteerd systeem bevat mogelijks gevoelige informatie. Dit kan worden gestolen.
6. Command and control: Geïnfecteerde apparaten moeten opdrachten kunnen ontvangen van de hacker. Hiervoor communiceren ze met een zogenaamde Command and Control (C&C) server. Deze communicatie kan geïmplementeerd worden over verscheidene protocollen. Dit helpt zo ook weer om detectie te ontwijken.
7. Lateral movement: Dit slaat op het verspreiden naar andere systemen op een privénetwerk waar men toegang tot heeft gekregen door het infecteren van een apparaat.

8. Gevolgen: Het geïnfecteerde apparaat wordt misbruikt om de hacker zijn uiteindelijke doel te bereiken. Dit kan gaan van het mijnen van cryptomunten tot het kapotmaken van het apparaat.

Communicatie tussen apparaten is een van de kernkenmerken van IoT. Zwakke hardware en mogelijks slechte netwerkverbindingen vereisen dat protocollen bepaalde karakteriseren bevatten, zoals een lage overhead en het kunnen garanderen dat berichten aankomen bij de ontvanger door middel van Quality of Service (QoS) opties. In deze thesis werden drie applicatielaagprotocollen onderzocht:

- MQTT: In dit protocol communiceren cliënten door zich te abonneren op, en berichten te publiceren naar, onderwerpen. De netwerkstructuur volgt een client-servermodel. De server associeert cliënten hun staat met `ClientID`s, en niet met hun inloggegevens. Dit kan session hijacking tot gevolg hebben. Verder kan men zich abonneren op alle onderwerpen, met behulp van wildcards, om zo berichten af te luisteren.
- UPnP/SSDP: Dit protocol laat apparaten toe om, zonder menselijke interventie, elkaar rechtstreeks te ontdekken, configureren en aan te sturen. De ontdekking van apparaten gebeurt over UDP. Dit kan misbruikt worden in een reflectieaanval om grote hoeveelheden trafiek te generen. Verder is er geen authenticatie voorhanden om acties uit te voeren, aangezien alles automatisch moet kunnen verlopen. Afhankelijk van de actie in kwestie kan dit ook een beveiligingsrisico vormen.
- CoAP: Dit protocol is ontworpen met IoT, alsook compatibiliteit met het web, in gedachten. Het lijkt dus op HTTP, maar is veel compacter zodat er geen fragmentatie van pakketten benodigd is. Doordat het over UDP werkt, is er, buiten de applicatielaag, ook een laag voor connecties en QoS voorhanden. Het gebruik van UDP maakt echter weer reflectieaanvallen mogelijk.

A.3 Honeypots

Honeypots imiteren productiesystemen om zo hackers te lokken, in de hoop dat ze hun technieken en intenties prijsgeven. Een honeypot bestaat uit twee delen: de decoy en de captor.

De decoy is wat de aandacht van de hackers moet trekken. Het moet dus opvallen, maar ook geloofwaardig uitzien en aantrekkelijk zijn om mee te interageren. Honeypots die services imiteren kunnen gevonden worden door middel van netwerkscans. Het is echter ook mogelijk om ze te "adverteren" door hun informatie te publiceren, bijvoorbeeld in een online document met foutieve permissies of op fora. Indien een honeypot niet geloofwaardig is, zullen hackers dit opmerken en vroegtijdig hun aanval stopzetten. Het verbeteren van dit kenmerk vergt echter extra werk. De geloofwaardigheid wordt dus best afgestemd op het doel van de honeypot. Dit heeft tot gevolg dat er een onderscheid tussen low en high fidelity projecten wordt gemaakt. Een low fidelity honeypot is meestal een software her-implementatie van een systeem. High fidelity honeypots zijn daarentegen gebaseerd op echte systemen. Verscheidene karakteristieken zoals mogelijke acties, kost, complexiteit, risico tot misbruik en onderhoud nemen toe indien de geloofwaardigheid toeneemt. Schaalbaarheid en geloofwaardigheid hebben daarentegen een omgekeerd evenredig verband.

De captor is alle achterliggende infrastructuur. Het moet ervoor zorgen dat het systeem niet misbruikt wordt, alsook dat gegevens correct verzameld en beschermd worden. Dit omvat het opslaan van ruwe, ongefilterde data op een plaats die niet toegankelijk is vanuit de honeypot en het correct samenvoegen van verschillende honeypots hun data. Verder moeten er ook genoeg gegevens verzameld worden om zo gebeurtenissen naderhand correct te kunnen reconstrueren. Hoog geloofwaardige honeypots geven hackers meer vrijheid. Dit laat toe om meer informatie te verzamelen, maar kan er ook toe leiden dat de honeypot misbruikt wordt voor illegale activiteiten. Dit moet zo veel mogelijk voorkomen worden, bijvoorbeeld door een firewall in te stellen.

Het doel van deze thesis is om een geloofwaardige honeypot te bouwen op basis van consumentenapparatuur. Om de captor te implementeren moeten delen van het apparaat gevirtualiseerd worden. Dit noemt firmware re-hosting. We bestuderen vier technieken:

- Full device proxy: Deze aanpak kan gezien worden als een gesofisticeerde proxy die hackers indirecte toegang verleent tot echte apparaten. De proxy probeert toepasselijk te antwoorden op netwerkverzoeken. Indien er geen antwoord geweten is, wordt het verzoek doorgestuurd naar een achterliggend apparaat. Verzoeken worden echter eerst gescand om misbruik te voorkomen. Een nadeel van deze aanpak is dat de apparatuur de schaalbaarheid, kost en onderhoud negatief beïnvloedt.

- Peripheral forwarding: De hardware en firmware van IoT-apparaten zijn sterk gekoppeld. De kans is dus hoog dat generieke, virtuele randapparatuurimplementaties een crash veroorzaken in een firmware-emulatie. Deze techniek probeert dit op te lossen. De firmware wordt gevirtualiseerd. Alle verzoeken om de hardware te gebruiken worden doorgestuurd naar een echt apparaat. Opnieuw beïnvloed de benodigde apparatuur deze aanpak. Verder is het verbinden van de emulatie met de hardware niet enkel technisch complex, maar introduceert het ook een significante vertraging in alle interacties.
- Virtual peripheral modelling: Deze techniek probeert automatisch virtuele randapparatuurimplementaties te genereren zodat de firmware kan geëmuleerd worden zonder een achterliggend apparaat. Sommige implementaties van deze techniek hebben een echt apparaat nodig om de juiste werking ervan te leren, terwijl anderen het proberen af te leiden uit de werking van de firmware. Het resultaat is een software-implementatie van de randapparatuur. Het gebruik ervan benodigd dus geen hardware meer. Het nadeel van deze aanpak is dat de geloofwaardigheid van de gegenereerde randapparatuur niet te vergelijken valt met echte hardware.
- Full system re-hosting: In plaats van de randapparatuur aan de geëmuleerde firmware aan te passen, wordt in deze techniek de firmware gepatcht. Aanpassingen omvatten, onder andere, het aanmaken van virtuele netwerkinterfaces, het vervangen van NVRAM-opslag met een speciaal gemaakte library en het aanpassen van het `init`-proces. Het leren van de benodigde aanpassingen is een iteratief proces. Deze techniek benodigd geen hardware. De geloofwaardigheid is echter enkel oppervlakkig door de vele aanpassingen benodigd om de emulatie te laten werken.

We maken onze eigen vergelijking van de vier technieken in tabel A.1, gebaseerd op honeypot kenmerken, om te weten te komen hoe toepasselijk ze zijn bij het bouwen van een honeypot. De conclusie is dat de geloofwaardigheid van de apparaten gevirtualiseerd met behulp van peripheral forwarding en virtual peripheral modelling niet voldoende is. De andere twee technieken hebben beiden hun eigen voor- en nadelen.

Criteria	Full device proxy	Peripheral forw.	Virt. peripheral	Full re-hosting
Kost	↑	↑	↑ (genereren) / ↓ (gebruik)	↓
Complexiteit	∼	↑	↑ (genereren) / ↓ (gebruik)	↓
Risico	↑	↓	Randapparatuur afhankelijk	↑
Geloofwaardig (gebruik)	↑	↓	↓	∼
Geloofwaardig (data)	↓	↑	↑	↑
Schaalbaarheid	∼	↓	↓ (genereren) / ↑ (gebruik)	↑
Onderhoud	↑	↑	↓	↓

Table A.1: Relatieve vergelijking van verschillende firmware re-hosting technieken op basis van de karakteristieken van een honeypot. Er wordt verondersteld dat de onderzoeker maar een beperkt aantal apparaten bezit. De symbolen ↑, ∼, ↓ betekenen hoog, gemiddeld en laag respectievelijk.

A.4 Implementatie

We bouwen een netwerk uit honeypot apparaten met behulp van Docker. Dit laat ons toe om het honeypotnetwerk (decoy) van de infrastructuur (captor) te scheiden. Alle componenten zijn virtueel. Het systeem kan dan ook simpel in een VM of op een VPS worden geïnstalleerd. Wij hosten onze implementatie op de cloud providers AWS en OVH.

Om manuele interacties uit te lokken, adverteren we de honeypots op fora, websites gelijkaardig aan Pastebin en door Google Docs® documenten publiek te maken. Er wordt vier keer per dag gepubliceerd om de kans tot interactie te verhogen. Elke advertentie heeft een uniek wachtwoord voor mensen om mee in te loggen op de honeypots. Zo weten we welke bezoeken het gevolg zijn van het lezen van een advertentie.

Het honeypotnetwerk bevat een IP-camera en een router die worden geëmuleerd met behulp van FirmAE, een framework dat de full system re-hosting techniek gebruikt. De emulatie met FirmAE is niet gegarandeerd om te slagen. Het vinden van werkende firmwarebestanden nam dan ook wat tijd in beslag. De camera draait een Telnet- en webserver. De router draait daarentegen een UPnP/SSDP-service. Deze services worden aan het internet blootgesteld. Voor de Telnet-service plaatsen we ook een proxyserver. Buiten het loggen van de sessies, laat deze ook toe om in te loggen met verschillende wachtwoorden. Dit kan gebruikt worden om niet enkel te bestuderen welke wachtwoorden in het algemeen gebruikt worden, maar ook om unieke wachtwoorden toe te laten die enkel via de advertenties geweten kunnen worden. Enkele aanpassingen moesten worden gemaakt om de web- en UPnP-services geloofwaardig te laten lijken. De NVRAM van een IoT-apparaat bevat normaal zijn instellingen. Bij full system re-hosting is deze echter initieel leeg. Als gevolg ontbreken er waardes in de antwoorden van applicaties. De vervanging van de NVRAM moest dus manueel aangevuld worden. Ook dit was moeizaam werk. Verder werd de webserver aangepast zodat het een nep videobeeld toont. Ten laatste moesten bestanden en processen van het FirmAE framework verborgen worden om detectie van de honeypot tegen te gaan.

Het netwerk bevat ook enkele gesimuleerde MQTT-cliënten, alsook een MQTT-server. MQTT werd gekozen omdat het het populairst is van de drie bestudeerde protocollen. Dit zijn "low fidelity" honeypots. Ze worden niet rechtstreeks aan het internet blootgesteld. Hun doel is namelijk het verspreiden door het privénetwerk uit te lokken. Er is geen authenticatie en encryptie ingesteld op de MQTT-server. Verder laat het, onder andere, session hijacking aanvallen en het afluisteren van trafiek toe.

De instrumentatie op de host bevat ook nog volgende services:

- Internettap: Met behulp van `tcpdump` wordt alle trafiek dat door het honeypotnetwerk gaat opgeslagen.
- Misbruik detectie: Deze service reset de honeypot zodra het misbruik detecteert. Dit is zodat één aanval niet alle toekomstige datacollectie kan beïnvloeden door bijvoorbeeld bestanden te verwijderen of het systeem te zwaar te belasten.
- Firewall: De firewall zorgt ervoor dat enkel de honeypot services kunnen worden blootgesteld aan het internet. Ook plaatst het een zwaar maximumlimiet op de bandbreedte. Zo wordt de bijdrage van een geïnfecteerde honeypot aan een aanval gelimiteerd, en worden de emulaties niet overspoeld met trafiek.

A.5 Analyse

Deze analyse omvat de data die werd verzameld over een periode van twee maanden. De honeypots werden echter langer online gehouden in het geval dat een opmerkelijke gebeurtenis zou worden gedetecteerd.

De verhouding tussen de hoeveelheden aan netwerktrafiek per protocol komt overeen met eerder onderzoek. Zo kreeg Telnet de meeste trafiek met tussen de 10^3 en 10^4 binnenkomende pakketten per dag op een honeypot. HTTP staat tweede met een factor 10 minder binnenkomende pakketten op gemiddelde dagen. HTTP en SSDP trafiek piekten echter van tijd tot tijd boven Telnet uit. Deze pieken komen overeen met misbruik van de honeypot. Verdere analyse van de trafiek toont dat door malware gegenereerde trafiek soms zo geconstrueerd is dat het als een ander protocol wordt herkend. Dit kan detectie belemmeren.

Tabel A.2 toont Telnet-sessie statistieken. We maken enkele observaties:

1. Het aantal unieke IP-adressen is minder dan het aantal succesvolle logins. De reden is dat aanvallen soms sporadisch falen en ze op een later moment opnieuw worden gestart. De meest waarschijnlijke reden hiervoor is netwerk timeouts als gevolg van een zwakke netwerkconnecties en/of hardware.
2. De duur van sessies in combinatie met het aantal getypte woorden impliceert dat aanvallen automatisch gebeuren.
3. Sessies bevatten ook IP-adressen. Deze worden gebruikt om malwarebestanden van te downloaden. Merk op dat het aantal unieke IPs minder is dan het aantal unieke malwarebestanden. Uniekheid van bestanden wordt bepaald door hun hashwaardes. Bepaalde IP-adressen hosten dus meerdere bestanden. We veronderstellen dat dit is doordat hackers hun malware soms aanpassen.

4. Het aantal verzamelde malwarebestanden is ettelijke keren groter dan het aantal unieke bestanden. Met andere woorden, men probeerde de honeypots meerdere malen met dezelfde malware te infecteren. Aangezien het herstarten van een apparaat genoeg kan zijn om malware te verwijderen, speculeren we dat hackers op deze manier proberen te verzekeren dat zo veel mogelijk apparaten geïnfecteerd zijn op een gegeven moment.

Statistiek	AWS EC2	OVH VPS	Totaal
Unieke IPs	3632	3084	5957
Succesvolle logins	20010	15800	35810
Sessies met unieke instructies	734	863	1365
Mediane sessieduur (seconden)	11,7	13,4	12,6
Mediane woorden per sessie	105	107	106
Unieke IPs vermeld in sessies	75	113	141
Verzamelde malwarebestanden	4392	4320	8712
Verzamelde unieke malwarebestanden	236	306	420

Table A.2: Statistieken gehaald uit interacties gelogd door de Telnet proxy, per honeypot.

Om de algemene werking van aanvallen gestuurd over Telnet-sessies te bestuderen, clusterden we ze met K-means. Het resultaat wordt getoond in figuur A.1. De instructies binnen een sessie kunnen in de volgende stappen worden opgedeeld: configuratie, informatie verzamelen, malware bestand downloaden, executie en schoonmaak. Merk op dat enkele clusters in de figuur dicht bijeen liggen. De reden is dat ze enkele stappen gelijkaardig aanpakken.

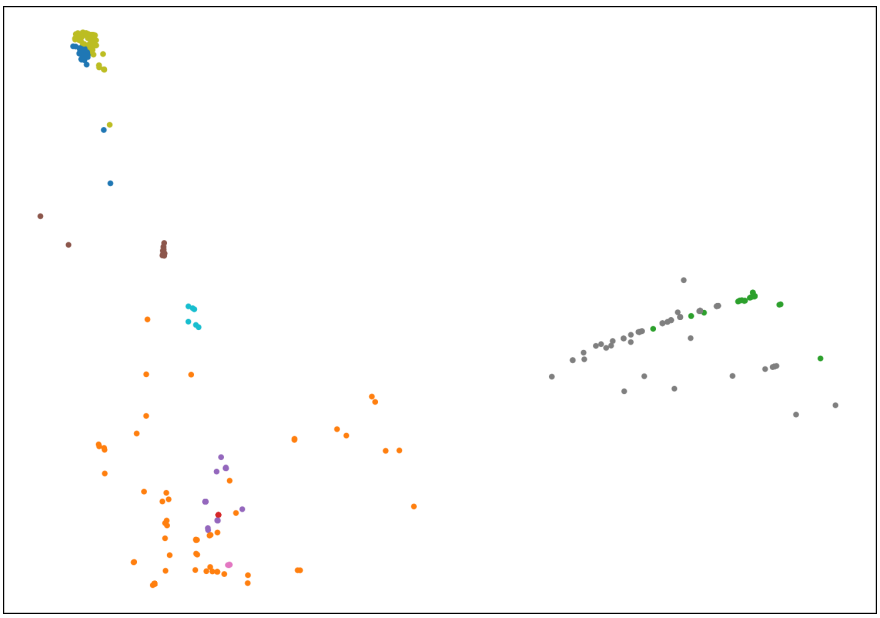


Figure A.1: Grafische representatie van de clustering van unieke sessies gebaseerd op hun geassocieerde instructies. Clusters worden aangeduid met kleuren. Elke sessie is een bolletje.

De verzamelde unieke malwarebestanden zijn hoofdzakelijk gecompileerd voor de MIPS (32-bit LSB) architectuur. Dit komt overeen met de architectuur van de IP-camera-emulatie. Meestal worden dus enkel toepasselijke bestanden gedownload in een aanval. We uploaden de unieke MIPS-bestanden naar VirusTotal, een website die meerdere virusscanners publiek aggregeert. Uit de resultaten van VirusTotal blijken de bestanden toe te behoren aan twee grote malwarefamilies: Mirai en Gafgyt. Deze hun programmacode is publiek beschikbaar. Dit maakt het gemakkelijk om nieuwe IoT-malwarecampagnes te starten. Het is dan ook geen verrassing dat, van de 255 unieke MIPS-bestanden, $20,4\%$ nog niet eerder gekend was op VirusTotal. Verder observeren we dat $41,6\%$ van de bestanden met UPX gecomprimeerd zijn. Dit obfusceert de binaire code. Opvallend is echter dat enkel 18 van de gecomprimeerde bestanden ook aangepast zijn geworden zodat decomprimeren niet meer mogelijk is. We veronderstellen dus dat het gebruik van UPX eerder is om bestanden kleiner te maken, dan de analyse te compliceren.

Binnenkomende HTTP-trafiek kan in drie groepen gesplitst worden. De eerste groep bestaat uit scans die zoeken naar beschikbare paden op de webserver. Hackers proberen detectie te omzeilen door populaire zoekmachines in de `Referer` HTTP-header te vermelden. Dit laat het lijken alsof de bezoeker de honeypot heeft gevonden via de zoekmachine. De tweede groep bestaat uit pogingen om de webserver te misbruiken. In de laatste groep zitten fatsoenlijke visites aan de webserver van de IP-camera. De duur van de bezoeken impliceert dat sommige manueel gedaan zijn. De bezoekers tonen interesse in de systeemopties en het videobeeld. Ze passen echter niets aan en verlaten de website vlak na het interageren met de video. We concluderen dat onze aanpassing om de video te kunnen tonen niet geloofwaardig genoeg is.

Buitengaande HTTP-trafiek bestaat daarentegen hoofdzakelijk uit websites die werden aangevallen nadat de honeypots waren geïnfecteerd, en IP-adressen om malwarebestanden van te downloaden. Het bevat ook verzoeken om een update te downloaden voor een van de services van de router. Het gebruik van consumentenfirmware betekent dat standaardservices zullen draaien op de honeypot. Hier moet aan gedacht worden tijdens de designfase om de integriteit van de data te waarborgen.

Het bestuderen van UPnP- en SSDP-trafiek toont dat deze service op twee manieren werd misbruikt. Enerzijds werden reflectieaanvallen gebruikt om grote hoeveelheden trafiek te genereren, en anderzijds probeerde men een poort te mappen door acties te misbruiken. Het tweede werkt echter niet omdat de UPnP-service zichzelf hiertegen beschermt. Afhankelijk van het doel van de honeypot is het dus van belang om firmware met gepaste services uit te kiezen.

Ten laatste analyseren we alle trafiek die door het honeypotnetwerk ging om te kijken of dat hackers hebben geprobeerd andere systemen, zoals de MQTT-simulaties, aan te vallen. Helaas is dit niet het geval.

A.6 Conclusie en toekomstig werk

Aan de hand van de verworven kennis kunnen de onderzoeksvragen als volgt beantwoord worden:

- Welke technieken en/of malware gebruiken hackers bij aanvallen waarbij IoT-apparaten betrokken zijn?
  De malwarebestanden vallen onder de families Mirai en Gafgyt. Indringen gebeurt door middel van standaardinloggegevens en publiek bekende kwetsbaarheden. De specifieke technieken, zoals het gebruik van UPX en het hernoemen van processen, zijn niet nieuw. De effectiviteit en proliferatie van IoT-malware is te wijten aan het gebrek van best practices.

- Slimme apparaten worden gemaakt om zeer specifieke problemen op te lossen. Ze verschillen dus veel van elkaar. Hoe passen hackers zich aan aan deze verschillen?
  Alhoewel er kleine verschillen zijn, zoals de architectuur van een apparaat en de inloggegevens benodigd om toegang te krijgen, proberen hackers hun malware zo generiek mogelijk te maken. Dit laat toe om zo veel mogelijk apparaten te infecteren. Het generieke besturingssysteem van Type 1 apparaten faciliteert dit.

- Proberen hackers zich door privénetwerken te verspreiden, en hoe goed lukt hen dat?
  We hebben geen pogingen hiertoe kunnen detecteren. Dit is verrassend aangezien connectiviteit een kernkenmerk is van IoT.

- Zijn state of the art virtualisatietechnieken bruikbaar om geloofwaardige honeypots te bouwen op basis van consumentenfirmware?
  Om alledaagse, geautomatiseerde aanvallen te verzamelen is firmware re-hosting niet aan te raden. Deze aanvallen zien namelijk geen verschil tussen een generieke firmware en een consumentenfirmware. Indien men menselijke interacties wil bestuderen, kunnen full device proxy en full system re-hosting mogelijks gebruikt worden, mits voldoende inspanning om de honeypot geloofwaardig te maken. In het algemeen zijn verbeteringen vereist om firmware re-hosting praktisch bruikbaar te maken.

Uit onze ervaringen geven we mee dat er zeker genoeg data verzameld moet worden. Ook moet er gedacht worden aan eventuele standaardservices van de firmware die data genereren. Verder is het belangrijk om de gevolgen van misbruik te beperken. Ten laatste, indien firmware re-hosting wordt gebruikt, merken we op dat het vinden van toepasselijke consumentenfirmware niet simpel is.

We geven volgende onderzoeksvragen mee voor toekomstig werk:

- Proberen hackers, indien ze manueel te werk gaan, zich door privénetwerken te verspreiden, en hoe goed lukt hen dat?
- Hoe kan firmware re-hosting verbeterd worden voor algemeen gebruik, alsook voor het bouwen van honeypots?
- Hoe kan het verwerven van firmware voor veiligheidsonderzoek verbeterd worden? Is het reverse engineeren van begeleidende mobiele apps een mogelijke aanpak? Hoe effectief is firmware re-hosting voor het emuleren van firmware die op deze manier werd vergaard?