Towards automated DDoS abuse protection using MUD device profiles

Master thesis

Caspar Schutijser

August 2018

Samenvatting

Onveilige Internet of Things-apparaten (IoT-apparaten) vormen een gevaar voor de stabiliteit van het Internet. Deze onveilige IoT-apparaten worden gebruikt om Distributed Denial of Service-aanvallen (DDoS-aanvallen) uit te voeren. De Manufacturer Usage Description (MUD) is een specificatie die wordt ontwikkeld in de Internet Engineering Task Force. Het doel van MUD is om netwerkbeheerders een stuk gereedschap aan te reiken waarmee de netwerktoegang van IoT-apparaten beperkt kan worden. MUD stelt een fabrikant in staat om de gewenste netwerktoegang van een apparaat te specificeren. Het netwerk kan dan de netwerktoegang van het apparaat beperken tot het strikt noodzakelijke, zodanig dat het apparaat zijn werkzaamheden kan uitvoeren.

In dit onderzoek wordt de toepasbaarheid van MUD voor het beveiligen van IoT-apparaten tegen hackpogingen en de bruikbaarheid in DDoS-aanvallen onderzocht. Een systeem waarmee MUD-profielen automatisch gegenereerd kunnen worden wordt ontworpen en geïmplementeerd. Vervolgens wordt gecontroleerd of de IoT-apparaten de werkzaamheden nog steeds correct uit kunnen voeren als het profiel wordt gehandhaafd. Verder wordt er een theoretische analyse uitgevoerd. Het doel van deze analyse is tweeledig. Ten eerste zal onderzocht worden of het handhaven van een profiel kan voorkomen dat een IoT-apparaat wordt gehackt. Ten tweede zal worden onderzocht of een IoT-apparaat kan worden misbruikt in een DDoS-aanval, mocht het toch gehackt worden.

De gekozen benadering lijkt goed te werken voor *specific-purpose* (in tegenstelling tot *general-purpose*) IoT-apparaten. Verder maken de gegenereerde profielen het inderdaad moeilijker om een IoT-apparaat te compromitteren. Voor het reduceren van de slagkracht van IoT-apparaten in DDoS-aanvallen is het echter wel noodzakelijk om bandbreedtebeperkingen op te leggen, zeker gezien het feit dat steeds meer services op cloudplatformen worden gedraaid.

Abstract

Insecure Internet of Things (IoT) devices are posing a threat to the stability of the Internet. These insecure IoT devices are used to perform Distributed Denial of Service (DDoS) attacks. The Manufacturer Usage Description (MUD) is a work in progress specification in the Internet Engineering Task Force. The MUD attempts to provide network operators with a tool to limit the network access of IoT devices. The MUD allows a vendor to specify the network access requirements of a device. The network is then able to restrict the network access of the device to the absolute minimum that is required to let the device carry out its functions.

The applicability of the MUD in protecting a device against hacking attempts and usability in DDoS attacks is examined in this research. A system to automatically generate MUD profiles is designed and implemented. It is then verified whether the IoT devices are still able to function properly once the profile is enforced. Furthermore, a theoretical analysis is performed. The goal of the analysis is twofold. First, we will verify whether enforcing a profile prevents an IoT device from being hacked. Second, we will verify whether an IoT device can be misused in a DDoS attack if it were hacked anyway.

For specific-purpose (as opposed to general-purpose) IoT devices, the approach taken to generating MUD profiles appears to work well. Furthermore, the generated profiles do indeed make it harder to compromise an IoT device. However, in order to make IoT devices less useful in DDoS attacks once they are compromised, it is recommended to apply rate limiting, especially as more services are moving to cloud platforms.

Acknowledgements

I (or should I say "we"?) would first like to thank my supervisors, Elmer Lastdrager from SIDN Labs and Roland van Rijswijk-Deij from University of Twente, for their guidance during this project. Thanks for the useful feedback, questions and ideas. Furthermore, I would like to thank my colleagues at SIDN Labs, I enjoyed my time as a student at SIDN Labs. Additionally, I would like to thank the DACS group at the University of Twente for providing me with a nice place to work on Thursday and Friday, and for the many chats.

Furthermore, I would like to explicitly thank Elmer Lastdrager, Roland van Rijswijk-Deij, Jelte Jansen and Moritz Muller for reading earlier versions of my thesis. Their feedback was very valuable to me.

Finally, I would like to thank my parents, my brother, my sister and my friends for their support. Without your support, I would not have been able to do this.

Contents

Chapter 1

Introduction

In the past, most devices were not connected to the Internet, either because the Internet did not exist yet or it was too expensive to connect them. These days, that is not the case any more and as such, it is more common to connect devices to the Internet. This phenomenon is sometimes called the Internet of Things (IoT).

In an in-home setting, customers are usually unaware of the fact that (IoT) devices must be managed. This means that security updates often are not installed and that the default settings of the devices are not changed [49]. As such, the adoption of IoT devices results in an enormous number of Internet-connected devices that can be exploited with relative ease. The Mirai botnet exploited this situation and created a botnet of IoT devices that was used to perform Distributed Denial of Service (DDoS) attacks against a number of companies and important infrastructure, including Dyn DNS [6, 11]. The scale of disruption caused by Mirai was considered an existential threat to the Internet [26]. Other IoT botnets emerged besides Mirai, such as Reaper [34].

The Manufacturer Usage Description (MUD) [37] is a work in progress specification by the Operations and Management Area Working Group (opsawg) working group [31] at the Internet Engineering Task Force (IETF). The idea behind this specification is that, once an IoT device connects to a network, the device informs the network about what network resources it needs to function properly. This information is contained in a *MUD profile*. It describes the intended network activity of a device in a whitelist-based manner. Since the whitelist is supposed to be exhaustive, this means that access to any other network resource can be denied without impeding the functionality of the device. As such, this should be an effective way of restricting the network access of an IoT device. As a consequence, this may reduce the attack surface of the device and as such may make the device more secure.

The goal of the research documented in this thesis is to evaluate MUD profiles; specifically, to evaluate how useful MUD profiles are to prevent an IoT device from being hacked and from being misused in DDoS attacks. However, the MUD specification is not finished yet, let alone implemented on devices. Despite these barriers, it would be interesting to investigate MUD. Therefore, our goal is to generate MUD profiles automatically. Those generated MUD profiles are necessary to carry out the research, but generated MUD profiles are potentially useful to protect IoT devices that do not support MUD as well (under the assumption that they are not infected yet). In order to generate a MUD profile, it is necessary to determine what kind of network access a device requires. Furthermore, in order to evaluate whether a MUD profile is suitable for protecting an IoT device from being hacked, it is necessary to know how IoT devices were hacked in the past. As such, it is useful to investigate the characteristics of earlier attacks.

This research was carried out at Stichting Internet Domeinregistratie Nederland (SIDN) [51]. SIDN is the organization responsible for managing the .nl top-level domain. SIDN attempts to address the problem of insecure IoT devices being used in DDoS attacks with a project called Security and Privacy for In-home Networks (SPIN) [52]. SPIN is software that is intended to run on the routers of home networks. Currently, the software visualizes the network activity of IoT devices and the user is able to block certain traffic. The evaluation of MUD was carried out in the context of the SPIN project.

1.1 Research Questions

The goal of the research is to evaluate the applicability of MUD in the context of protecting IoT devices against hacking attempts and being misused in DDoS attacks. However, MUD as a specification is still a work in progress and as such, no devices currently on the market implement MUD. In order to be able to evaluate MUD despite this fact, MUD profiles will be automatically generated. The automatic generation of MUD profiles will stay relevant once the MUD specification is finalized, for instance to limit the network access of IoT devices that do not support MUD. This results in the following main question of the final project:

> To what extent can automatically generated MUD profiles be used to prevent IoT devices from being hacked and/or from being misused in DDoS attacks?

To answer the main question, the following questions will be answered first:

RQ1
: What information is needed to generate a MUD profile of an IoT device?

RQ2
: Are IoT devices able to function properly once generated MUD profiles are enforced?

RQ3
: Does enforcing the generated MUD profile prevent IoT devices from being hacked?

RQ4
: If an IoT device were hacked anyway, does enforcing a MUD profile prevent IoT devices from being misused in (for instance) a DDoS attack?

1.2 Structure

The remainder of this thesis is structured as follows. Chapter 2 provides background to this research and related work, Chapter 3 describes an architecture devised to generate and enforce profiles, Chapter 4 describes the prototype which implements the devised architecture, and Chapter 5 evaluates the implemented prototype. Finally, Chapter 6 summarizes the results and provides conclusions. Appendix A provides additional details regarding the implementation considerations of the prototype.

Chapter 2

Background and Related Work

This chapter provides information on a number of topics related to this research. The goal is to provide some background and to show what kind of research has already been done which will be useful in this work.

As attacks such as Mirai showed, there are a number of IoT devices on the market that are easy to hack and misuse in attacks. The insecurity of IoT devices is discussed in Section 2.1. In this research, the plan is to evaluate the usefulness of the Manufacturer Usage Description (MUD). However, the MUD specification (which is described in Section 2.2) is still a work in progress. As a consequence, no implementations of MUD exist yet, both in IoT devices and in the network infrastructure that would support enforcing such a profile. Despite the fact that MUD is not yet finished, it would be interesting to be able to evaluate the usefulness of MUD. In order to do that, two things are needed that do not yet exist: profiles for IoT devices and a way to enforce such profiles. In order to be able to create a profile for an IoT device, it must first be clear what information a profile actually consists of. Furthermore, it is necessary to know how this information can be gathered. A review of existing literature on this topic can be found in Section 2.3. Furthermore, to assess the effectiveness of enforcing profiles against hacking attempts, it is necessary to know about the characteristics of earlier attacks. Section 2.4 will give an overview of information in this area. Finally, Section 2.5 will address other attempts at generating MUD profiles.

2.1 Insecurity of IoT Devices

Before discussing how to protect IoT devices, we first need to discuss the state of IoT security and the security practices of the IoT industry. Unfortunately, poor security and disregard for best practices are the rule rather than the exception in the IoT market. This is shown by Antonakakis et al. [11], who describe how the Mirai botnet grew and infected other devices. The authors note that an important factor in the success of Mirai was the fact that security best practices are not followed by most vendors in the IoT industry. For instance, many devices are shipped with default passwords. This made it feasible to log in to hundreds of thousands of devices with a dictionary attack (using a small list of known default usernames and passwords). Furthermore, IoT devices are shipped with a number of ports opened by default, accessible to anyone, even though that is unnecessary for the device to function.

Due to the way most new IoT products are developed, it is often hard or impossible for the vendors to patch vulnerabilities or to support the product for the entire lifetime of the product. This situation is aptly described by Bruce Schneier [48]. Chipset vendors do not take the time to build a proper architecture that can be supported for a long time. Rather, new chipsets are rushed to market and once the chipset has been released, work begins on a new chipset. Instead of documenting the hardware and releasing open source drivers, it is common practice to use closed source drivers, also known as *binary blobs*. Such drivers often only work with a specific software version, like the 4.4 branch of the Linux kernel. The fact that the driver only works with a specific version of the software means that it is difficult to support (i.e., patch) the software once that specific version reaches the end-of-life (EOL) state. Note that this situation is not limited to the IoT market; for instance, the "smartphone" market suffers from the same problems, particularly in the case of Android phones [18].

There are early signs that the industry is starting to understand that it is necessary to keep Internet-connected devices supported for a longer period of time. The Civil Infrastructure Platform (CIP) [1] is a project hosted by the Linux Foundation that receives support from a number of key industry players such as Hitachi and Siemens [4]. One of the goals of the project is to create a *super long-term* supported kernel [17] that should be maintained for 20 years or even longer. However, this project requires long-term commitments from the industry and it remains to be seen whether that will be the case. Furthermore, before this project brings about the desired change, it must first be incorporated into products by the manufacturers, something that does not happen overnight. As such, this effort will not contribute to improving the situation in the short term.

In conclusion, the fact that most IoT devices are unpatched and insecure is a fact that will remain unchanged in the short term. Therefore, it is necessary to investigate how to protect IoT devices against outside threats. One possible solution is limiting the network access of the devices. In the long term, the development process of IoT device manufacturers should change such that it becomes viable to properly support the software for the entire lifetime of the products. Efforts such as the *super long-term* supported Linux kernels could help in that respect.

2.2 The Manufacturer Usage Description

The Manufacturer Usage Description (MUD) [37] is a work in progress specification currently being written by the opsawg IETF working group. In summary, the idea behind MUD is that once an IoT device connects to a network, the device tells the network what kind of network access it needs to perform its functions. For instance, some devices may only need to access the printer on the local network and the update service of the manufacturer to do their job. As such, the network access of the device can be limited to those two network resources without impeding the functionality of the device, which potentially improves the protection of the IoT device against unauthorized access and the consequences thereof, such as being part of a DDoS attack.

MUD is specifically targeted towards IoT devices, as opposed to general-purpose computing systems. The reasoning behind that decision is that IoT devices supposedly have a well-defined function and as such, it should be fairly straightforward for the manufacturer to enumerate the network resources they need. Therefore, it is considered feasible to create a whitelist that can be enforced successfully without interfering with normal usage. This is much harder for general-purpose computing systems, as the manufacturer does not know beforehand how the device will be used.

When analyzing these statements a bit further, it becomes clear in what cases MUD is supposed to be applicable (at least according to the vision of the authors of MUD). Devices that have a specific and fairly static function fall within the bounds; devices on which all kinds of apps can be installed (which brings all kinds of network access requirements as well) are not within bounds. Examples named in the specification that fall within bounds are light bulbs and printers. Examples of devices not covered by MUD are "smartphones" or "smart" TVs. Those are devices that lean more towards being a general-purpose device.

Since the specification is still in a work in progress state, there are currently no devices that implement this specification. One of the authors of the specification did say that he knows of two software implementations of MUD [36]. However, those implementations are not publicly available yet.

According to the authors of the MUD specification, it is the sole responsibility of the manufacturer to create an appropriate MUD profile for a device; the manufacturer is considered a trusted party. The reason for that is that the manufacturer is the only party that can correctly determine what network resources a device needs and what resources it does not need. However, since the manufacturer is fully trusted in this model, the possibility exists that manufacturers will create MUD profiles in which the device is allowed to do more than absolutely necessary to perform the functions of the device. Something similar happens in the "smartphone" market, where applications request more permissions than strictly necessary [22]. On the other hand, if the manufacturer does not want to place any restrictions on what network resources the device can access, the manufacturer may choose to not create a MUD profile at all. Possibly, manufacturers could be forced to implement proper MUD profiles, for instance by government regulations.

The specification mentions some security considerations. For instance, what is preventing a device from acting like it is another device in order to get more permissions on the network? The authors have some ideas on addressing this issue, for instance using IEEE 802.1AR certificates [5]. Using this standard, "A Secure Device Identifier (DevID) is cryptographically bound to a device and [it] supports authentication of the device's identity" [30]. This requires the vendor to embed additional hardware in the device. Note that security considerations regarding the transport and authenticity of MUD profiles are not related to the research questions. As such, those considerations are out of scope for this research and not discussed any further.

2.3 Determining Device Network Access Requirements

The problem of determining what kind of network access a device requires can be approached from multiple angles. Those angles are described in this section.

Attempting to create a profile of the behavior of a device such that certain traffic can be flagged is not a new concept. In fact, that is one of the methods to perform intrusion detection. A survey conducted by Sabahi et al. [47] shows that when applying intrusion detection, one way to process the information is to apply profile based anomaly detection. When applying anomaly detection, it is necessary to "define a region representing normal behavior" [15]. As such, there first is a training phase, during which a profile of the normal behavior is built, followed by a testing phase, during which the profile is used to classify new data [42]. Often, defining such a region is not an easy task for various reasons. For instance, it may be hard to define a model that includes all normal behavior. Furthermore, the normal behavior may change over time.

RFC 2722 [14] outlines a way of looking at network traffic. Network traffic is described as a collection of *flows*. A stream of packets is considered to be part of a particular flow if a set of *attributes* match. In the case of Internet traffic, such attributes typically include the source and destination IP addresses, the protocol used on the transport layer and transport layer port numbers (if applicable). This specific set of attributes is also known as the *five-tuple*. Additional attributes may be stored. For instance, attributes that are frequently stored are timestamps that indicate when the first and last packet of a flow were observed. Furthermore, it is possible to keep track of the number of packets and bytes that were exchanged. "Network entities" that observe packets are called *meters*. A typical example of a meter is a router. Each meter stores flow information in so-called *flow tables*. That way, the information can be queried later. An implementation of a system that collects flow information is NetFlow [16]. NetFlow is typically used in corporate networks. With NetFlow, network traffic is usually sampled for performance reasons.

Flow records contain IP addresses, not the domain names that were used to look up the IP addresses. In certain applications, the domain name belonging to an IP address in a flow record is more interesting than the IP address itself. After all, when a user or an application connects to a server, a DNS lookup is performed to obtain the IP address for a given domain name. Therefore, if the operator of the domain name changes the IP address of the domain name, a future flow will contain a different IP address, even though the user is connecting to the same service. To overcome these problems, Bermudez et al. [13] annotate flow records with domain names. This is done by inspecting DNS answer packets and associating the resulting IP addresses to the IP addresses found in the flow records. Note that the reverse DNS lookup of an IP address often does not provide useful information on which specific domain or subdomain was accessed. Therefore, just performing a reverse DNS lookup is not sufficient.

With Software-Defined Networking (SDN), the so-called *control plane* is detached from the *data plane* [12]. Effectively, this means that a network switch just forwards packets according to some rules (*flows*). Those flows are installed by a *controller*, an external system. If a packet arrives that does not have an applicable flow, the packet is sent to the controller. The controller can then inspect the packet and make a decision as to what needs to happen with the packet (for instance, the controller can opt to create a new flow in the switch). Flows can match a packet based on certain properties of a packet, such as source/destination MAC address, source/destination IP address, source/destination application level port and some other properties.

Mehdi et al. [38] bring SDN to the home network. They use OpenFlow to analyze the network connections that are set up. With OpenFlow, a packet that does not match one of the installed flows is sent to the controller. Mehdi et al. leverage this by not installing any flows into the router. As such, every time a new connection is set up, the controller is informed and gets to decide whether the connection should be allowed, in which case two flows are installed, or whether the connection should be dropped. This way, it is possible to inspect every connection while keeping the number of packets that need to be analyzed by the controller low.

In the area of Internet of Things, Habibi et al. [27] provide a solution specifically tailored towards IoT devices. The proposed system attempts to create a profile for each device, mainly consisting of "a whitelist of all the destinations that the device can legitimately contact in order to perform its functions." All traffic is considered benign, unless the destination is present on the VirusTotal blacklist, in which case the traffic is blocked. The system continuously evaluates new destinations and adds them to the whitelist as necessary. According to the authors, this is a “practical and low-overhead” approach.

2.4 Characteristics of Earlier Attacks

In this section, we describe literature that provides information on the characteristics of earlier attacks and hacking attempts. Such information is useful in order to understand how to protect IoT devices from being hacked and misused in attacks. This allows us to validate the generated MUD profiles, which in turn allows us to answer Research Questions 3 and 4.

Khattak et al. [32] provide an analysis of botnets. Specifically, it discusses how to detect botnets and how to defend against them. It provides a taxonomy of botnets in general, not about one botnet specifically. According to Pa et al. [41], telnet daemons are (still) present on a significant number of devices and used to build botnets. Kishore [10] similarly notes that telnet (and sometimes SSH) is used to gain access to devices in order to add them to a botnet.

There is also literature available about specific botnets, such as Mirai. Mirai is a botnet that infected IoT devices and used those devices to perform DDoS attacks. Mirai is interesting in particular because it was able to take Dyn DNS offline [6, 11]. Fortunately, the behavior of Mirai is well-documented. For instance, the propagation strategy is described by Kolias et al. [33]. An infected device scans the Internet for other vulnerable devices. Mirai probablistically attempts to connect to either TCP port 23 or port 2323. If it succeeds in setting up a connection, it tries to log in to the device using a small list of known usernames and passwords (shipped by default on the devices). Once infected, the devices were used for DDoS attacks. Mirai performed application layer attacks, volumetric attacks and TCP state exhaustion attacks, as noted by Antonakakis et al. [11]. Furthermore, it is noted that the IP address of the targeted device is encoded in the TCP sequence number of the probe packet. By doing so, the scanning process can be made stateless which makes it more efficient. This information aids the detection of Mirai traffic.

Another botnet, Reaper or IoT_reaper, has been discovered by Netlab 360 [2, 3]. Reaper propagates by using known (but unpatched) vulnerabilities. The developer(s) of Reaper actively add new exploits to their toolkit as new vulnerabilities become public. The infected devices connect to a number of known IP addresses and domains, for instance to fetch commands or to share information with the botnet operators. This should make it straightforward to detect Reaper botnet activity. So far, the botnet has not been used for an attack but it is clear that a new botnet is being built and it may just be a matter of time before it will be used in a DDoS attacks or other unwanted activities. Another example of a botnet that is likely to exploit known vulnerabilities is the Satori botnet [7]. After the publication of a new buffer overflow vulnerability in the uc-httpd web server [40], the botnet started scanning TCP ports 80 and 8000, port numbers that are often used for web servers.

Once a device has been compromised and added to a botnet, the attacker often continues interacting with the hacked device. For example, the attacker may want to perform a DDoS attack or update the malware installed on the device. In other words, the device needs to be controlled by the attacker. This is called *command and control* [21]. There are different ways attackers interact with the devices in their botnets. Those ways are often categorized as (1) a centralized architecture, with the infrastructure controlled by the attacker, or (2) a distributed architecture, using peer-to-peer networks [29]. In the past, centralized botnets often used Internet Relay Chat (IRC) to communicate with their devices. These days, centralized botnets often communicate using HTTP or a custom protocol on top of TCP. For instance, the Satori botnet reports port scan results to a server running at a specific IP address and port [7].

Attackers build botnets to carry out DDoS attacks, for example. A DDoS attack can be carried out in many ways [55]. For example, the attacker can instruct the devices to flood a victim with ICMP, UDP or TCP packets with the goal of saturating the Internet connection of the victim. Instead of using the devices to attack the victim directly, it is also possible to carry out a amplification attack. When carrying out a amplification attack, an attacker sends a small packet to a server - often a server running a UDP-based service such as memcache, DNS or NTP [9, 19] - soliciting a big response. This small packet contains a spoofed IP source address, the address of the intended victim. As a result, the big response will be sent to the victim rather than the hacked device, contributing to the DDoS. Unfortunately, IP address spoofing remains a usable strategy as long as many Internet Service Providers do not implement BCP 38 [23]. Besides amplification attacks, the hacked devices can also target the victim directly. Possible attacks include various types of flooding, such as SYN flooding or ICMP flooding [43].

One of the ways IoT botnets are investigated is by deploying honeypots. Honeypots [46] are systems that are used to observe what attackers are doing. Usually, honeypots are systems that are easy to log in to, similar to vulnerable IoT devices. Such systems are easy to log in to for instance due to the use of passwords that are easy to guess. Once the attacker logged in successfully, the attacker's activity is carefully monitored. This allows the operator of the honeypot to learn about the activities of the botnets. Possibly, the botnets attempt to infect the honeypot with malicious software that would add the honeypot to the botnet. In this case, the operator of the honeypot would obtain a copy of that malware which allows the malware to be investigated. Using honeypots, Pa et al. were able to determine that a majority of the investigated botnet families support UDP flooding and TCP flooding as methods to perform DDoS attacks [41].

The information presented in this section provides an insight into the approaches taken by attackers. This is useful in the this research as this improved understanding makes it possible to verify whether the developed measures actually improve the safety of the IoT devices.

2.5 Other Attempts at Generating MUD Profiles

During the course of this research, a paper was published by Hamza et al. named *Clear as MUD: Generating, Validating and Applying IoT Behaviorial Profiles* [28]. In this paper, the authors attempt to generate MUD profiles by first creating a pcap of the network traffic of a device. The pcap is then fed to a tool called mudgee which generates a MUD profile for the device. Rather than verifying whether the MUD profile helps against hacking attempts, the authors "checks its [the generated MUD profiles] compatibility with a given organizational policy". As it happens, the approach taken to generate the MUD profile is quite similar to the approach taken in this research. The fact that those researchers independently designed a similar system may indicate that the approach taken is the logical first choice.

Chapter 3

Approach

The goal of this research is to evaluate MUD and its applicability in protecting IoT devices against hacking attempts and usability in DDoS attacks. A key element of evaluating MUD is the need for device profiles. However, at the start of this research, a system able to create such profiles did not exist yet. Therefore, it was necessary to create a system that can somehow create such profiles. Collecting information necessary to create profiles and constructing profiles by hand does not scale. Therefore, the goal is to automate this process. In order to reach the above stated goals, the following requirements are defined:

Requirement 1
The system must collect information which can be used to generate MUD profiles.

Requirement 2
During the collection phase, the system must be able to process live network traffic, as well as recorded network traffic (from a pcap file, for instance).

Requirement 3
The system must be able to enforce a generated MUD profile in order to limit the network access of an IoT device.

Requirement 4
All processing (i.e., the collection, generation and enforcement of a profile) must be performed on the router of the in-home network.

From the requirements, a number of activities that the system needs to perform become clear. Those activities are depicted in Figure 3.1. The activities outlined in the figure are described in more detail in the remainder of this chapter.

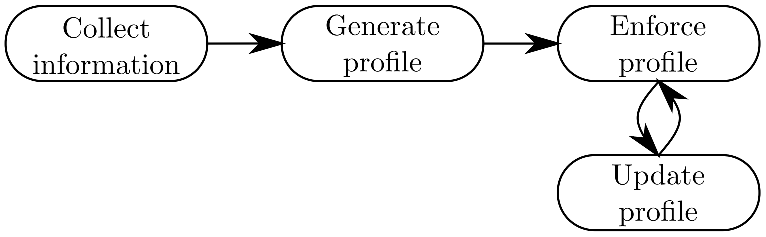


Figure 3.1: Schematic overview of the activities of the system.

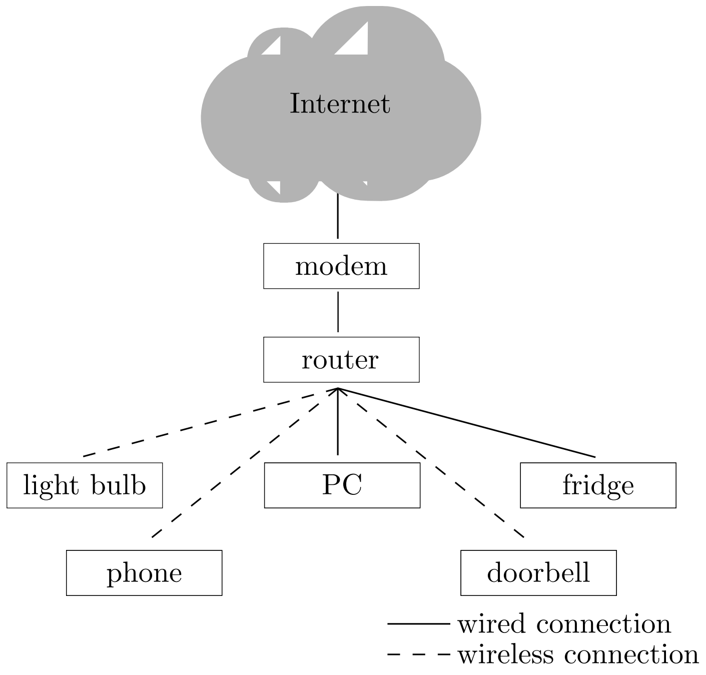


Figure 3.2: Schematic overview of a typical home network.

3.1 Collecting Information

The first step in generating a profile is actually collecting the necessary information. From a high level, a stream of packets will be observed and relevant information will be extracted and stored. The remainder of this section will describe these steps in more detail.

In Chapter 2, methods of determining what kind of network access a device needs were outlined. Such information can be used to create a profile of a device's network activity. In this research, flow records (see Section 2.3) were used to characterize the traffic. For a number of reasons, flow records are very suitable for this research. For instance, flow records contain the type of information that is necessary to build profiles of network activity of a device. Furthermore, compared to other methods such as deep packet inspection, flow records are an efficient way of keeping track of network activity. It is efficient in terms of the required processing power, as well as storage requirements. This is an advantage since the network traffic will need to be analyzed on the home router. The home router usually is constrained in terms of processing power and storage capacity.

Information about the network activity of a device can only be collected from a device that is on the path from the device to the Internet. Compared to a corporate network, the typical home network infrastructure is usually not very sophisticated (see Figure 3.2): all network devices are in the same *broadcast domain* and sometimes, all devices are directly connected to the home router (either via Wi-Fi or via a network cable, possibly with Ethernet switches in between). As such, the home router is on the path to the Internet for all devices on the network, which makes it a suitable spot for collecting information. Another device that is also on the path to the Internet for all devices is the modem (although, sometimes the modem and router are integrated into one device). However, the modem is tasked with decoding signals from the wire into zeros and ones and vice versa. Specifically, the modem is not concerned with the interpretation of the information that is transferred with the stream of bits. Therefore, it is not practical to inspect IP traffic at this level.

The collected data must be stored somewhere for later use. The network infrastructure of a home network usually consists of just the modem and the router (sometimes those two devices are even integrated). Not adding another device to the infrastructure lowers the barrier for consumers to actually install such a device in their network. As such, it is preferable to store the collected data on the device itself, i.e. on the router.

For these reasons, we decided to use the home router for data collection and storage in this research.

3.1.1 Processing the Packets on the Wire

During the collection process, a stream of packets is observed. In order to collect flow records, it is not necessary to perform deep packet inspection. This has a number of advantages. For instance, deep packet inspection comes with privacy concerns. Additionally, performing deep packet inspection on all packets would not be practical due to the processing power restraints. Furthermore, the use of encryption reduces the usefulness of deep packet inspection [50]. As such, only a subset of the available information will be used.

When looking at the OSI model, information from layer 2 upwards is available. For each packet, the following information is inspected (categorized by OSI model layer) and stored:

Layer 2
The Ethernet MAC addresses in each packet.

Layer 3
Source and destination addresses in the headers of the IPv4 or IPv6 packets, and the transport layer protocol (examples: TCP or UDP). (if applicable)

Layer 4
The port numbers of the TCP and UDP headers, and the size of the payload in bytes. (if applicable)

The information described above can be used to reconstruct flow records that describe the network activity of a device. Information that is not necessary to create flow records is not stored. Notably, the payload of TCP and UDP packets is not stored. Furthermore, IP header fields such as the time to live and the checksum or the TCP sequence and acknowledgement numbers are not stored, again because they are not necessary to reconstruct flow records.

Besides collecting basic information as described above, additional information is gathered by performing deeper inspection on certain types of packets. To be more specific, this is the case for ARP (and its IPv6 counterpart named NDP), TCP, and DNS.

ARP and NDP
MAC addresses (OSI layer 2 addresses) can be used to uniquely identify a device while a device may have multiple IP addresses (OSI layer 3 addresses). Furthermore, the layer 3 addresses may change over time, for instance because they are often assigned dynamically. As such, it is necessary to create a mapping between layer 2 addresses and layer 3 addresses.

However, it is not sufficient to just store all combinations of layer 2 addresses and layer 3 addresses that appear on the network interface. We will demonstrate this with an example. Host A resides in the `192.168.8.0/24` subnet. The IP address of host A is `192.168.8.123`, and the gateway of the subnet is `192.168.8.1`. If host A wants to communicate with host B (`192.168.8.20`) which resides in the subnet, host A can send the packet directly to `192.168.8.20` using Ethernet. This means that the layer 2 destination address will contain the layer 2 address of host B, and the layer 3 destination address will contain the layer 3 address of host B. However, when host A wants to communicate with `212.114.98.233`, a host outside the subnet, the packets must be *routed* by the gateway. In this case, the layer 2 destination address will contain the layer 2 address of the gateway while the layer 3 destination address will equal `212.114.98.233`. If we would store all combinations of layer 2 and layer 3 addresses, the gateway would appear to have a lot of layer 3 addresses while that is not true. This shows that it is not sufficient to store all combinations of layer 2 and layer 3 addresses that appear on the network interface; rather, it must be verified whether a layer 3 address belongs to a device that is on the local network. When processing live traffic, information about the network (such as the netmask) is available and could be used to make a distinction between layer 3 addresses that are inside the subnet and addresses that are outside the subnet. However, when processing recorded traffic (pcap files, for example), such information is not available.

Fortunately, this information can be extracted from the Address Resolution Protocol (ARP) and Neighbor Discovery Protocol (NDP) protocols. ARP is used to find the MAC address for a given IPv4 address while NDP is used similarly for IPv6 addresses. This is done by broadcasting an ARP or NDP request into the network. All devices that reside in the same *broadcast domain* or subnet receive such a packet and are able to respond. When a device receives an ARP or NDP request and the IP address configured on the network interface equals the IP address requested in the packet, the device will respond with a reply. Therefore, it is necessary to extract this information from the network traffic by inspecting ARP and NDP traffic.

TCP

Besides inspecting the ARP and NDP packets, the Transmission Control Protocol (TCP) deserves special attention as well. When a TCP connection is initiated by a client, the client sends a TCP packet with the `SYN` flag enabled to a server. If the server decides to accept the connection, the server replies with a packet with both the `SYN` and `ACK` flag enabled. Finally, the client responds with a packet in which the `ACK` flag is enabled. From this point onwards, the client and the server are able to exchange data. This is known as the three-way handshake. The presence of the `SYN` flag can be used to deduce which host initiated the connection. This bit of information is stored for later reference. Why we will need this information will become clear in Section 3.2.2.

DNS

The final protocol that receives more attention is the Domain Name System (DNS) at OSI model layer 7. The DNS is used, among other things, to obtain an IP address for a given domain name. This is useful because users do not like to remember IP addresses. Furthermore, using a domain name rather than an IP address unties a service from the location at which it is hosted. As such, when a device connects with an IP address, that specific IP address is not very interesting on its own when it was obtained using the DNS. The device may connect to a different IP address in the future if the IP address for the domain name is changed by the service's operator. Therefore, DNS packets are inspected more deeply[1]. Specifically, DNS packets that contain an answer (one or multiple

[1]This is the only case where deep packet inspection is performed.

IP addresses) to an earlier asked question (a domain name) are inspected. This is done such that an IP address can later be mapped back to a domain name. This is similar to the approach taken by Bermudez et al. [13].

Sufficient information has been collected once one is certain all kinds of data has been measured. In order to reach this point, it is a good idea to (a) make sure all features of the device have been used as each feature can expose different network access requirements, and (b) leave the device running for a minimum amount of time (24 hours, for instance). This way, network traffic generated by periodic activities are captured as well. An example of such a periodic activity is an automatic update check that is performed at specific time intervals. To illustrate how these considerations work out in practice, imagine an Internet-connected light bulb that can be controlled through an application on a phone. The light bulb can be switched on or off, the color and the brightness can be changed, and possibly a time schedule can be set up. Using the different features may use different API's and therefore requires different network access. Furthermore, the device may check for software updates every 24 hours. This feature again will expose different network access requirements.

3.2 Generating a Profile

Now we will explain how a profile is generated. A profile can be generated once sufficient information has been collected (see the previous section). A profile consists of a whitelist of destinations a device is allowed to contact. Additionally, it also contains a whitelist of remote systems that are allowed to initiate contact with the device. The flow information captured during the collection process (described in the previous section) will be used to generate those whitelists. This section describes the process behind generating those profiles.

3.2.1 Selecting Relevant Flows

The flow records that were collected in the previous step have been persisted by the collection program. These persisted flow records contains information about the network traffic of all devices in the network. In order to generate a profile for a specific device, the relevant information needs to be selected from the collected data.

Each device is connected to the network using a network interface (be it an Ethernet interface or Wi-Fi interface). Those network interfaces can be uniquely identified using a MAC address. As such, the network activity of a device is tied to this MAC address. Therefore, to generate a profile for a specific device, all flows matching a certain MAC address should be selected.

An alternative but inferior option is matching flows based on the IP addresses. While it certainly is possible to select the flows matching a certain IP address, IP addresses are usually allocated dynamically to a device and as such are not a robust way to attribute traffic to a specific device. This is especially the case when measurements are performed over a longer period of time. This again highlights why it is useful to use the information obtained from ARP (and its IPv6 counterpart NDP) to determine which MAC addresses belong to devices on the local network.

3.2.2 Direction

Given the flow records, it is known which hosts communicated and which protocols were used. However, as an example, in the case of a TCP [45] connection that was set up successfully, there will be two flow records: a flow record with traffic from host A to host B, and a flow record with traffic from host B to host A. This is because packets flowed in both directions. For generation of the profile, it is necessary to know which side of the connection initiated the connection. That is necessary because the server port is the relevant piece of information, while the source port is often chosen at random by the client and as such should not be used in the profile. In the case of TCP, determining which side initiated the connection is straightforward: as was described in Section 3.1.1, information about the connection setup is embedded in the protocol header. As such, it is straightforward to deduce this information from just inspecting the packet headers.

In the case of UDP [44], this is not as straightforward. UDP is a stateless protocol and as such is not aware of the concept of “connections”. This does not mean that the client-server model, used as an example earlier, is not used with UDP. When using DNS, for instance, the DNS resolver library (the client) sends a DNS query to a DNS server. A DNS server is typically listening on UDP port 53 while the client port is usually chosen at random. As such, with UDP it is also the case that the server port is the important piece of information that is to be used in the profile while the source port is of little relevance.

3.3 Enforcing a Profile

Once a profile has been generated, the profile can be enforced. This means that the network access of the device will be restricted to the whitelists specified in the profile. Before we can enforce a profile, we need to make a number of decisions. For instance, we must decide in which location in the network the profile will be enforced. Furthermore, we will need to decide how the profile is actually implemented.

The profile will be enforced at the home router. The reasons are similar to the reasons the home router is responsible for inspection of the packets during the collection phase (see Section 3.1).

The profile can be enforced in a couple of different ways. At first sight, a straightforward approach appears to be to generate firewall rules and install them into the firewall. The home router presumably already has firewall software installed and enabled, or it is possible to install and enable a firewall. However, this method has a drawback. The profile consists, among other things, of a list of domain names. Domain names can be resolved to IP addresses. However, DNS records have a TTL associated with them, which means that the returned result will not stay valid indefinitely. As such, it is not sufficient to look up a domain name and use the resulting IP addresses in a firewall rule; if the IP address for a domain name changes, the user will connect to the wrong IP address.

Therefore, another approach is necessary. One approach to solving the problem of expiring DNS records is to keep track of the DNS traffic during the time the profile is enforced. When a DNS query and answer is observed, the query can be looked up in the whitelist. If the domain name is present in the whitelist, the resulting IP addresses can be whitelisted in the firewall. If the domain name is not in the whitelist, nothing needs to be done. However, if the DNS traffic is inspected passively, a race condition may occur in the following sequence of events: (1) A DNS answer is delivered to the device while the DNS answer has not been processed yet by the application responsible for inspecting the DNS traffic. As such, the firewall rule allowing the traffic to the destination has not been added yet. (2) The device attempts to contact the destination. Since the firewall rule allowing the traffic to the destination has not been added yet, the device will fail to contact the destination and will observe failure. Depending on the behavior of the application running on the device, this sequence of events will lead to a temporary or permanent failure. Either way, this race condition should be avoided. Therefore, it may be necessary to delay the DNS response until the firewall rule has been added.

An even better approach is to keep track of the DNS traffic, but not just as a passive observer. Rather, position yourself in the network stack and when a DNS answer arrives, look up whether the domain name is in the whitelist and if it is, somehow make sure traffic to the IP addresses is allowed and only once that has been arranged, the DNS answer will be sent back to the device that performed the DNS lookup. This will prevent the race condition that exists in the first solution.

Now that the issue regarding DNS has been addressed, we turn our attention to another problem. Most firewalls are OSI layer 3 firewalls. However, profiles are generated for a specific MAC address. A MAC address is an OSI layer 2 address. Therefore, a firewall cannot work with MAC addresses right away. To be specific: when a packet arrives from a network interface into the firewall, it is able to observe from which MAC address it came by inspecting the Ethernet header. However, it will not know which MAC address a packet will be delivered to since the MAC address corresponding to the IP address is looked up at a later stage in the ARP or NDP table. Despite those problems, it would be nice to use a firewall for enforcing the profiles. A solution to this problem is to look up the IP addresses belonging to the MAC address upon enforcing the profile.

Looking up the IP addresses belonging to the MAC address upon enforcing the profile comes with a disadvantage, however. If the IP address of a device changes (either legitimately or illegitimately), the device is able to bypass the imposed rules. This disadvantage could be mitigated by disallowing any traffic from or to local IP addresses that are not explicitly used by a profile. A stronger solution which immediately addresses the problem of devices spoofing their MAC address is to use something like IEEE 802.1AR [5], a solution also mentioned by the authors of the MUD specification. In order for this to be used, the devices need to support that standard.

3.4 Updating a Profile

This section describes why it potentially would be necessary to update a profile. Furthermore, it is described how a profile would be updated.

Once a profile has been generated and is being enforced, it may be necessary to update the profile of a device. This is necessary when the network access patterns of the device have changed. There are two reasons for this to happen. (1) The user may have changed their behavior. For instance, the user may have started using a feature that was not used during the information collection period. (2) The IoT device may have received a software update. A new version of the software running on a device may introduce new feature or change existing features. In either case, the device may attempt to access a network resource it does not have access to. As a result, it will experience failure. The first case can be prevented by making sure the collected data used to generate the profile is adequate. Guidelines on generating profiles can be found in Section 3.1.1.

If it is indeed deemed necessary to update the profile of a device, it is necessary to collect new information about the network activity of the device. This means that the profile that is currently enforced needs to be “unenforced”. After all, if the old profile is determined to be insufficient, the device must be able to contact that are not in the whitelist of the current profile. Effectively, the entire sequence of activities (see Figure 3.1) - collecting information, generating a profile, and enforcing a profile - has to be performed again to create an updated profile. Since the activity of updating a profile consists of steps that were already described previously, it is not necessary to write additional code in order to support this feature.

Chapter 4

Prototype

This chapter describes how the architecture described in Chapter 3 was implemented as a prototype. Building a prototype of the proposed system is beneficial. For instance, it can be used to show that the proposed architecture can be used to successfully generate and enforce profiles for IoT devices. This will assist in answering Research Question 2.

The prototype was built in the context of the SPIN project. Therefore, we will first describe the relevant parts of the architecture of the SPIN software. Furthermore, we will describe how the implemented prototype fits in the architecture of SPIN. Finally, we will describe the components that the prototype consists of.

4.1 The Valibox and SPIN

The SPIN software runs on the Valibox. The Valibox is a mini-router that runs a custom OpenWRT build. Originally, the goal of the Valibox was to provide a DNSSEC-validating recursive resolver for the in-home network. Nowadays, the Valibox also ships with the SPIN software as a prototype. The goal of SPIN is to protect the home network. It focuses on IoT devices and the security problems that result from using IoT devices.

The Valibox OpenWRT firmware typically runs on a GL-iNet device such as the GL-iNet

Figure 4.1: A GL-iNet AR150 running the Valibox OpenWRT firmware.

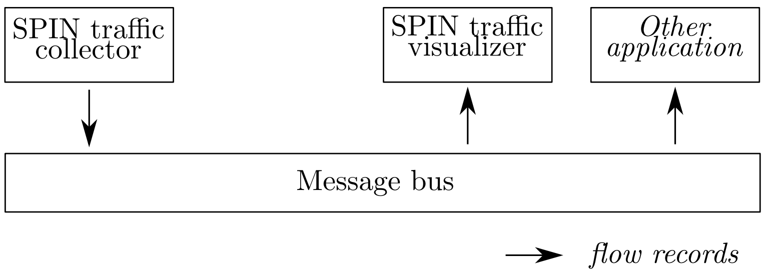


Figure 4.2: Schematic overview that shows how information about network traffic is distributed through the SPIN system.

AR150 [24] (see Figure 4.1). This mini-router has 64 megabytes of RAM and a MIPS 24Kc CPU running at 400 Megahertz. By default, it runs OpenWRT, a tiny Linux distribution geared towards embedded devices. OpenWRT is typically used on routers. The device has two Ethernet interfaces. One of the Ethernet interfaces is labelled *LAN* and can be used to connect the device to the local area network. The other Ethernet interface is labelled *WAN* and is intended to be used to connect the device to another device that can provide connectivity to the Internet. Furthermore, the device also has a Wi-Fi interface.

Currently, the SPIN software is able to visualize network traffic in a web application. It visualizes network traffic by depicting hosts as nodes; traffic between two hosts is visualized as an edge between the two nodes. Besides visualizing traffic, SPIN can be used to block certain traffic flows or to disconnect a device from the local network entirely.

The message bus is an important building block of the SPIN architecture, as is shown in Figure 4.2. The message bus is used to exchange information between components of SPIN. Using this model, information can be published onto the message bus by one or multiple *publishers* and the information can be used by one or multiple *consumers*. This model is used, amongst other things, to distribute information about network traffic that has been observed. The SPIN traffic collector observes network traffic flowing through the Valibox router and it publishes aggregated flow records onto the message bus. This information can be consumed by multiple applications. The SPIN traffic visualizer is an example of an application that consumes this information. Note that components in the system are not necessarily just a producer or consumer. For instance, the application that visualizes the network traffic can instruct another part of the SPIN system to disconnect a certain device. In this case, the traffic visualizer is not just a consumer, but it also publishes information.

4.2 Overview of the Prototype

The system described in Chapter 3 was implemented such that it leverages the architecture used by the SPIN project. Therefore, it uses the message bus and message formats as used by SPIN. As is shown in Figure 3.1, the proposed system needs to carry out four activities: (1) collect information; (2) generate a profile; (3) enforce a profile, and (4) update a profile. The prototype implements those activities. Figure 4.3 shows which components the prototype consists of. The remainder of the chapter will discuss the components of the prototype, guided by the four activities the prototype needs to perform. To make clear which component is being discussed, Figure 4.3 will be used throughout the chapter. For each component that is being discussed, the figure will be shown and the discussed component will be highlighted in the figure.

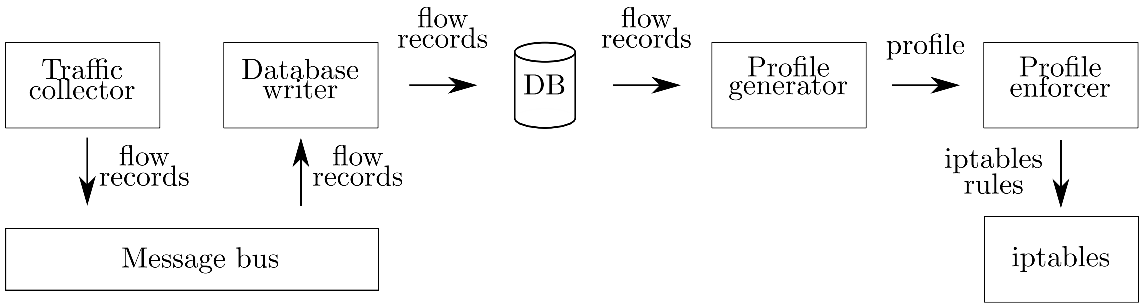


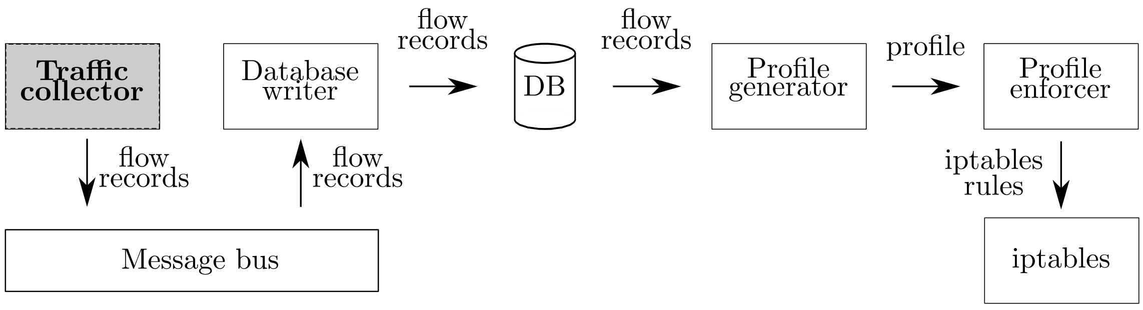
Figure 4.3: Schematic overview of the components necessary to generate and enforce MUD profiles.

4.3 Collecting Information

The first activity, collecting information, is taken care of by two separate components: the *Traffic collector* and the *Database writer* (see Figure 4.3). The traffic collector publishes the information onto the message bus. However, this information needs to be persisted such that information can be collected over a period of time and a profile can be generated later. Therefore, this activity consists of a second component as well, the *Database writer*. This component is responsible for storing the flow records in the database. It does so by subscribing to the message bus, and writing the flow records published by the traffic collector to a database. This component is described in Section 4.3.2.

The SPIN software already provides a traffic collector that publishes information about the network activity onto the message bus (see Figure 4.2). However, there are two reasons why the SPIN traffic collector is unsuitable for this research (in its current form). (1) The SPIN traffic collector is only able to observe live network traffic; in particular, it is not able to read a file containing network traffic that was collected earlier (like a pcap file). This is relevant because being able to use recorded traces makes it easier to test the software. Additionally, it is useful to be able to use pcaps that are provided by other people. (2) The SPIN traffic collector does not emit information on which side of a (TCP) connection initiated a connection. In order to generate accurate profiles, this information is necessary (see Section 3.2.2). Implementing this feature in the SPIN traffic collector was considered but ultimately this path was not pursued. Implementing it properly would probably be rather time-consuming. Furthermore, it could have crossed other efforts to improve SPIN in this area, resulting in duplicated or unnecessary work. Therefore, it was considered necessary to build a new implementation of the traffic collector. This new traffic collector is described in Section 4.3.1.

4.3.1 Traffic Collector



This section details our own traffic collector. Our own traffic collector can be used as an alternative to the SPIN traffic collector. In summary, it is able to parse the packets in a pcap file and produce output that is (more or less) compatible with the SPIN software. In addition to reading a pcap file, the program is also able to listen to a network interface to capture packets on a live network. The traffic collector is able to publish the extracted flow records onto the message bus in order to emulate the SPIN traffic collector.

The traffic collector uses `libpcap`, a common library used to either capture packets on a live network interface or read a pcap file which contains packets that were captured earlier. We decided to use this format and library because the pcap format is well supported by other tools. Examples of such tools include Wireshark, a tool used to inspect packet traces. Furthermore, the `libpcap` library is available on all popular general-purpose operating systems, which makes the program portable to other operating systems.

Handling the Packets

In order to collect flow records, the traffic collector inspects each packet that appears on the network or in the pcap file that is being read. As is shown below, a distinction is made based on the Ethernet type of each packet. The rationale for inspecting each of the packet types can be found in Section 3.1.1.

ARP
: If the packet is an ARP packet, it is verified whether the ARP packet is an ARP reply. An ARP reply is a response to an earlier broadcast ARP request, in which a device on the network asks which MAC address serves a certain IPv4 address. If that is the case, the MAC address and the IPv4 address are stored in a table. This way, it can be determined which IPv4 addresses belong to devices on the local network and which IPv4 addresses are not on the local network.

IPv4 and IPv6
: If the packet is an IPv4 or IPv6 packet, the “next protocol” field of those headers is examined:

  ICMPv6
  : (This can only happen if the packet is an IPv6 packet) In the case of an ICMPv6 packet, it is examined whether the ICMPv6 type of the packet is ND_NEIGHBOR_ADVERT. NDP is the IPv6 counterpart of ARP for IPv4.

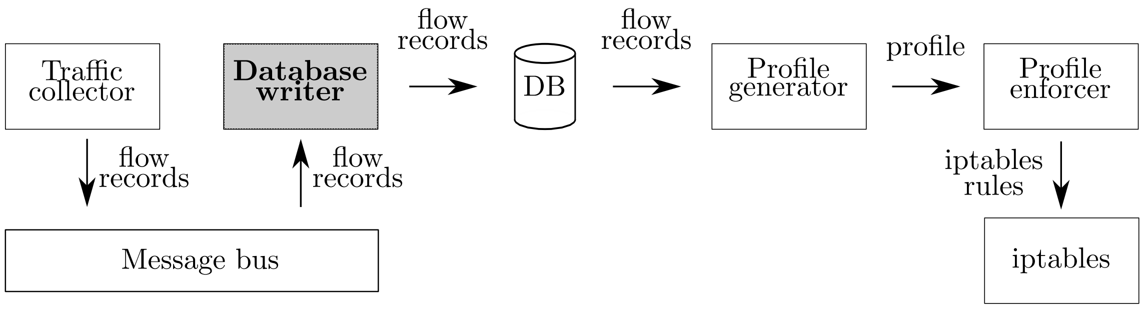
  TCP
  : The TCP port numbers are stored. Additionally, it is verified whether the packet is the initiation of a connection. This is the case when out of the `SYN` and `ACK` flag, only the `SYN` flag is set.

  UDP
  : The UDP port numbers are stored, similar to TCP.

Besides collecting flow records, it is also necessary to collect DNS responses. With the DNS responses, we are able to annotate IP addresses in the flow records with domain names. When either port number in a TCP or UDP packet equals 53, the packet could be a DNS packet so it is handed off to the function that attempts to parse DNS packets. To parse the DNS packets, the `ldns` library is used. It is verified whether the packet has any answer records. If that is the case, those are printed to the console. Note that the implementation is not aimed to be one of production quality; in particular, it is probably vulnerable to DNS cache poisoning. When more time is available, it is possible to mitigate such risks.

As was noted earlier (but repeated here to emphasise the fact), our traffic collector is able to deduce and export which side of a TCP connection initiated the connection while the traffic collector provided by SPIN in its current form is not able to do that.

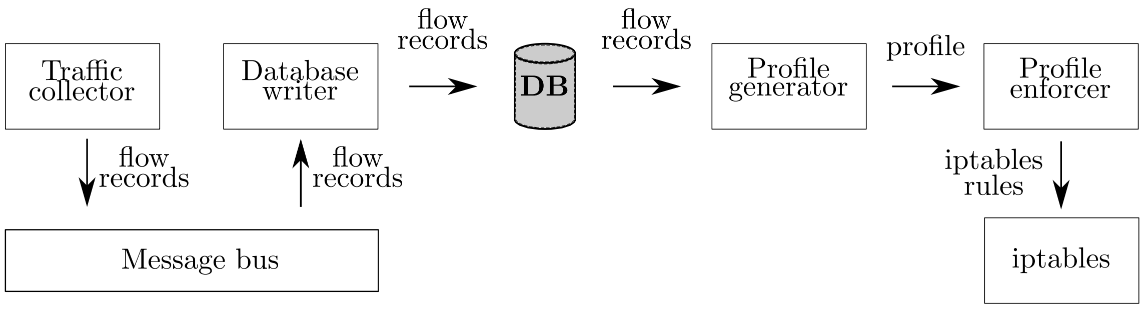
4.3.2 Database Writer



The database writer subscribes to the message bus and reads the data that is published by either our traffic collector or the SPIN traffic collector. The flow records are then stored in a SQLite database. The data model of the database can be found in Section 4.4.

Upon startup, the programs first make sure the SQL database exists; if it does not, it is created first. Then, it subscribes to the message bus. It will then processes the flow records one by one, as they appear on the message bus. For each flow record, it is first verified whether that flow already exists in the database. That is done by performing a `SELECT` query. The query verifies whether a flow with the same *attributes* (such as IP addresses, transport layer protocol and port numbers) already exists in the database. The flow must not be older than an hour, otherwise it is not considered "current". If a flow record already exists, an `UPDATE` query is executed to update the observed number of packets and bytes. If the flow record does not exist, an `INSERT` query is executed to insert the new flow in the database. The database currently has not been designed to achieve high performance. However, achieving high performance is out of scope as the goal is to build a prototype of the proposed system, not production grade software. More details on the data model of the database can be found in the next section.

4.4 The Database



A database is used by the database writer (see Section 4.3.2) to store flow records. The database software that is used to store the data is SQLite. This section first describes why SQL and SQLite were used, as opposed to alternatives. It then describes the data model of the database.

The flow records that are collected by the traffic collector are persisted by the database writer for later use. A number of options for storing the flow records are available. For instance, a possibility is to export and store NetFlow or IPFIX records. Another possibility is to use an SQL database. Yet another option is to use a custom file format.

Compared to SQL, NetFlow or IPFIX records are not very suitable for querying. However, querying the data is necessary when generating profiles. Additionally, storing the information we collect in the NetFlow or IPFIX formats does not provide any additional benefits. We consider that to be the case because we collect the same type of information that is collected when using NetFlow or IPFIX. With that being said and the querying argument in mind, we prefer SQL over NetFlow or IPFIX records.

When comparing SQL and a custom file format, we prefer to use SQL. Most importantly, there is a myriad of tools and libraries available to work with SQL and as such, it is straightforward to use SQL in applications. This makes it convenient for other developers to use the collected data. There are cases where using a custom file format is warranted, but in this case it would not provide much benefit while increasing the amount of work necessary to use the data in an application. Therefore, we decided that the collected flow records will be stored in an SQL database.

As the database software, we decided to use SQLite. SQLite is light-weight and has few dependencies, which makes it very suitable for use on resource-constrained devices such as the Valibox (see Section 4.1). Furthermore, SQLite is easy to use; using a database does not require setting up any users or granting users permissions. Yet, despite being a light-weight database, SQLite is powerful software that provides a relational database.

The database consists of a number of tables and a number of views. The most important table is the table named `flows`. It contains a representation of the flows observed by the traffic collector. The data model of the table can be found in Table 4.1. The second and last table is the `dns` table. The data model of this table can be found in Table 4.2.

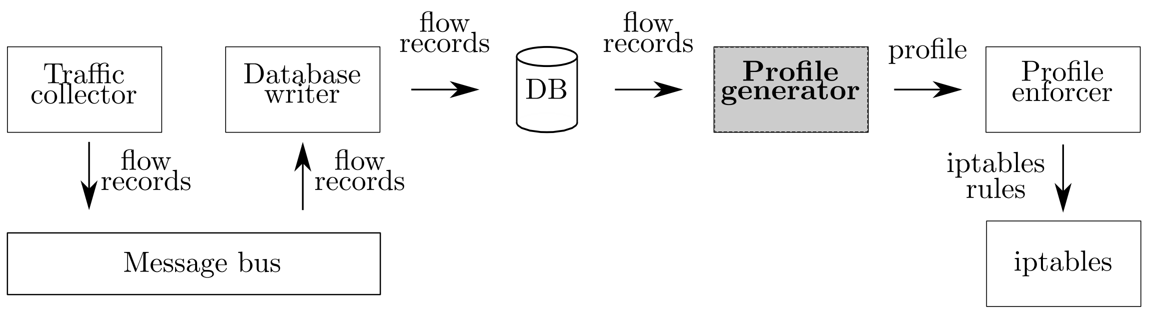
Note that the chosen type seems a bit odd at first for certain fields, for instance in the case of the IP address where the string type was chosen. An IP address could be more efficiently stored as an integer and the database could potentially perform input validation. However, the textual representation of an IP address is easier to use for developers. Some databases have a special type that can be used for IP addresses or MAC addresses [53] which allow for typing the IP address in a textual representation but use an efficient representation in the database. However, SQLite does not have such a type and therefore, a decision had to be made between efficiency and developer convenience. Since this code is part of a prototype and not production software, developer convenience was considered more important than efficiency and as such, the string type was used to store IP address.

Besides the tables, there is a view that combines the data from both tables into an annotated flows table which adds `domain_from` and `domain_to` fields. The fields equal `NULL` unless a domain name for a certain IP address is found in the `dns` table. The timestamp and TTL are taken into account.

Field name(s)	Field type	Description
`id`	Integer	ID that is automatically allocated.
`mac_from`, `mac_to`	String	The MAC address of the device the traffic originated from/to. Only available if the device was determined to be in the local network (see Section 3.1.1 for more details).
`ip_from`, `ip_to`	String	The IP addresses involved in the flow.
`ip_proto`	Integer	The transport layer protocol number as it appears in the IP protocol header. Examples: `6` (TCP), `17` (UDP).
`tcp_initiated`	Integer	Always `0` if `ip_proto` does not equal `6` (TCP). If `ip_proto` equals `6`, this field equals `1` if the TCP connection was initiated by the host specified in `ip_from`.
`port_from`, `port_to`	Integer	The transport layer protocol port numbers. Only valid if `ip_proto` equals `6` or `17`.
`packets`	Integer	Number of packets of this flow.
`bytes`	Integer	Number of bytes of this flow.
`first_timestamp`, `last_timestamp`	Integer	Timestamps in the UNIX timestamp format. `first_timestamp` signifies when the first packet of this flow was observed, `last_timestamp` signifies when the last packet of this flow was observed.

Table 4.1: Data model of the `flows` SQL table.

4.5 Generating a Profile



Field name(s)	Field type	Description
`id`	Integer	ID that is automatically allocated.
`domain`	String	The domain name of the lookup.
`ip`	String	The IP address that was returned for the domain that was looked up.
`timestamp`	Integer	UNIX timestamp at which the answer was observed.

Table 4.2: Data model of the `dns` SQL table.

The profile generator uses the database that contains flow records to generate a profile for a specific device. The data model of the database can be found in Section 4.4.

The program has three modes of operation, each of which is elaborated on below:

1. generate a profile for a specific MAC address.
2. generate a profile for a specific IP address;
3. generate profiles for all devices found in the database;

When generating a profile for a specific MAC address, only the flows related to that specific MAC address should be selected from the database. To that end, only the flows that have the selected MAC address in either the `mac_from` or `mac_to` field are of interest. With this selection in mind (which is implemented by adding a `WHERE` clause to each query which specifies the MAC address), a number of `SELECT` queries is performed. A number of distinctions is made when performing the `SELECT` queries:

- Does the traffic flow either from or to the specified MAC address?
- Is the transport layer protocol either TCP, UDP or something else?
- Was the connection initiated by the MAC address' device or was the connection initiated by the other side? This applies to the TCP and UDP cases. In the case of TCP, the value can be taken directly from the `tcp_initiated` field in the `flows` table in the database. In the UDP case, the program optionally guesses which side is the client or the server: when one of the port is lower than 1024, it is considered the server port. This heuristic stems from Unix. Historically, only the superuser (root) was able to bind to a port below 1024. These ports are sometimes called *privileged ports*. In the case of the DNS protocol, for instance, this guess proves to be useful as the DNS server listens on port 53. A better approach would be to keep track of this in the data collection program; however, since the necessary information is not exposed from the UDP headers (unlike the `SYN` flag in the TCP header), this would require more computing power.
- Is the remote service identified by an IP address or by a domain name?

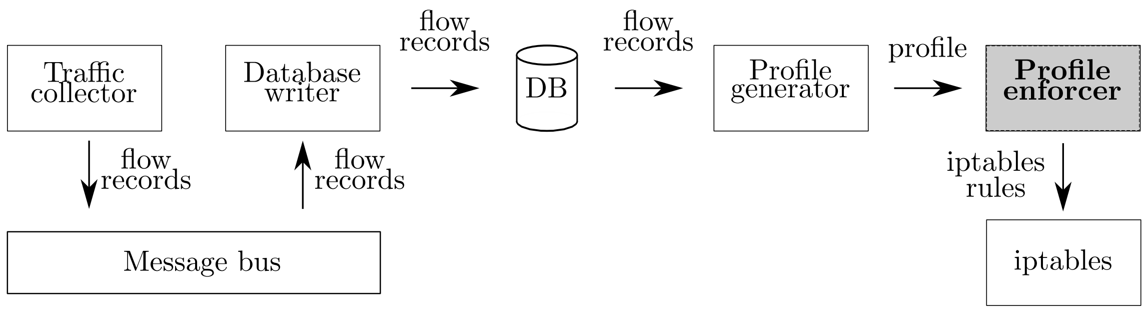
Generating a profile for a specific IP address is similar to generating a profile for a specific MAC address. However, instead of selecting a specific MAC address, a specific IP address is specified. Note that generating a profile for a specific IP address is usually not appropriate. It must be kept in mind that a device possibly has more than one IP address (an IPv4 address and an IPv6 address, for instance). Furthermore, IP addresses of a device may change. If possible, this feature should not be used.

Finally, the program is able to generate profiles for all devices found in the database. This is done by first querying for all MAC addresses found in the database. Then, for each MAC address, a profile is generated using the process described above.

A profile emitted by the profile generator contains the following information:

- The MAC address of the device.
- A list of hosts that the device is allowed to contact, separated by transport protocol (TCP, UDP). It also includes the port numbers.
- A list of hosts that are allowed to contact the device. Also in this case the transport protocols and port numbers are specified.

4.6 Enforcing a Profile



In order to enforce the profile generated in the previous step, the profile must be turned into firewall rules. Then, those rules have to be installed into the firewall. This is done by the *Profile enforcer*.

The output of the profile enforcer consists of lines that can be fed to a UNIX shell. Each line contains an invocation of either `iptables` or `ip6tables`. `iptables` operations that are performed are the flushing of the `FORWARD` chain, creation of new chains and insertion of rules in those new chains. The output of this program can be piped to a UNIX shell, which subsequently executes the resulting output and thereby imports the generated firewall rules into the actual firewall.

The firewall works with IP addresses while the profile is generated for a specific MAC address (see Section 3.3). This means that at the moment the profile is to be enforced, it is necessary to look up the IP addresses belonging to the MAC address. This is done by querying the *neighbor table* of the Linux kernel. For each of the IP addresses found in the table, the profile enforcer generates the same set of rules (with the exception that of course, `iptables` is invoked for IPv4 addresses and `ip6tables` is invoked for IPv6 addresses; additionally, there are no rules generated for `ip6tables` involving an IPv4 address and vice versa).

The iptables firewall is not able to work with domain names. Instead, it is necessary to specify firewall rules using IP addresses. Due to time constraints, we decided to look up the domain names at the time the rules are installed and use the IP addresses that were returned at that point in time. As described in Section 3.3, a disadvantage of this method is that a firewall rule will become outdated once the operator of the domain changes the IP address behind the domain name. However, given the fact that we are building a prototype, this disadvantage was considered acceptable.

The generated firewall rules are *stateful* rules. This means that the firewall, when evaluating a packet, does not just look at the packet under evaluation. Rather, any packets that may have been observed previously are also taken into account [25]. This is done by keeping track of packets that have been observed earlier during the same connection. In the case of TCP, the IP addresses and TCP port numbers are used to match packets to a certain connection. Additionally, the stateful firewall will keep track of the TCP state machine. In the case of connectionless protocols such as UDP or ICMP, there is no state machine to keep track of. Therefore, in the case of UDP, IP addresses and UDP port numbers are usually the only bits of information used to match packets to a connection. We will now illustrate how this works in practice with an example. Imagine that host A accesses the Internet through the stateful firewall. The firewall has a rule which allows outbound traffic to UDP port 53. Additionally, the firewall does not contain any firewall rules which allow inbound traffic. Host A sends a

UDP packet to host B, a host which is reachable by going through the firewall. The UDP packet has 53 as its destination port and 28433 as its source port. When the packet reaches the firewall, the firewall evaluates its rule set and finds that the packet is allowed to go through. Additionally, the firewall creates an entry in the *state table* which indicates that host A sent a packet to host B from port 28433 to 53. The packet is processed by host B and host B sends a reply. The source port of the reply packet equals 53, and the destination port equals 28433. The firewall verifies whether the combination of IP addresses, transport layer protocol and port numbers exists in the state table. Since that is the case, the packet will be allowed through. The packet is allowed through despite the fact that there is no firewall rule which allows any inbound connections.

4.6.1 Limitations of the Implemented Prototype

In the area of profile enforcement, the implemented prototype has a number of limitations. One of those limitations is the fact that domain names in profiles are looked up only upon profile enforcement (see the previous section). Additional limitations are discussed below.

Traffic on the local network

Traffic from or to the Internet flows through the firewall and as such is subject to the rules imposed by the firewall. However, traffic between two local devices does not go through the firewall since the devices are in the same *broadcast domain* and as such do not need a router to route packets for them; instead, the packets are sent directly to their neighbors using Ethernet. As such, an enforced profile will not restrict traffic that stays on the local network.

There are solutions to this problem. For instance, Mortier et al. [39] propose and implement a method to force devices to route their local traffic through the router as well. As a short and incomplete summary, the idea is to allocate a `/30` network for each device in the network. The device is assigned an IPv4 address in this `/30` and the router also is assigned an IPv4 address in this `/30`. Traffic between devices then needs to travel from one `/30` to the other and as such, it must be routed through the router. Therefore, the router is able to filter such traffic. This method breaks broadcast traffic; broadcast traffic stays inside a `/30` and as such does not reach other devices inside the network. Mortier et al. deal with this by rewriting the destination address of the broadcast traffic and making sure it is sent to the other device's `/30`s as well.

Due to time constraints, this method has not been implemented for this prototype and as such, local traffic is unaffected by the imposed rules. Only traffic from or to the Internet is subject to the enforced profile. To illustrate this, we will use the network from Figure 3.2 as an example. Traffic between the fridge and the Internet would be subject to the profile. However, traffic between the phone and the fridge would not be subject to the profile as the phone and the fridge both reside in the local network.

Portability of the generated firewall rules

The iptables rules that are generated by the profile enforcer are tailored towards the way iptables is used in OpenWRT. As such, in order to use the programs on a "normal" system, the profile generator needs to be adapted slightly.

4.7 Updating a Profile

The last activity, updating a profile, is not depicted in Figure 4.3. That is the case because, as was described in Section 3.4, the process of updating a profile is actually composed of steps that were already implemented previously. During the period in which the prototype was used, it has not been necessary to update the profile. As outlined in Section 3.4, there are two reasons for updating a profile: either the user behavior changed such that updating the profile becomes necessary, or a software update was installed onto the IoT device. However, neither of these events occurred during the testing phase. Therefore, it was not necessary to perform this step. However, since this activity consists of executing steps which were already implemented, updating a profile would be straightforward.

Chapter 5

Evaluation

With the prototype built, it is time to evaluate it. We evaluate the prototype in order to verify that the implemented system matches the expectations encoded in the Research Questions. The evaluation will be performed by defining criteria and verifying whether the prototype meets the criteria. The first step in evaluating the prototype is to determine what properties or statements actually need to be evaluated. The prototype was built based on the Research Questions. As such, the criteria will based on the Research Questions as well, in order to verify whether the prototype meets our expectations.

5.1 Defining Criteria

Research Questions 2, 3, and 4 relate to the properties of the resulting prototype. This is not the case for Research Question 1. Therefore, Research Questions 2, 3 and 4 are suitable for basing criteria on. Guided by those Research Questions, the following criteria are defined. Those criteria correspond directly to Research Questions 2, 3, and 4. We will verify whether the prototype meets those criteria.

Criterion 1
: The IoT device is able to function properly once a profile that has been generated for the device is enforced.

Criterion 2
: The enforced profile prevents the IoT device from being hacked.

Criterion 3
: The enforced profile prevents the IoT device from being misused in an attack if it were successfully hacked anyway.

5.2 Criteria Satisfaction

Now that the criteria have been defined, it will be determined for each of the criteria when they are met. All criteria have in common that in order to meet them, it is necessary to generate and enforce profiles for devices. In other words, all components of the prototype (as shown in Figure 4.3) will be exercised in order to get to the point where a generated profile can be enforced. Detailed instructions on how to use the prototype are found in Appendix A.2, so they are not repeated here. Instead, this section focuses on how to determine whether the criteria are met, which can be done if the prototype is used according to the provided instructions.

5.2.1 Criterion 1

Verifying whether criterion 1 is satisfied consists of two steps: (1) The profile should be generated and enforced (see Section A.2). (2) It should be verified that the device still functions properly. Step (2) consists of two parts: (2a) *Use the device and observe whether the device still functions according to expectations.* That is done by using the different features a device provides. For instance, in the case of a Philips Hue bridge, the tester could connect a phone and light bulbs to the bridge, turn on the light bulb, change the color of the light bulb, etc. (2b) *Analyze the network traffic that is generated while performing the tests.* In particular, we verify whether the traffic generated by the device is allowed by the profile. During this observation and while the profile is enforced, it may be the case that the device attempts to contact a domain that is not in the whitelist of the profile[1]. If that is the case, this possibly indicates that the profile is not complete. This needs to be investigated on a case by case basis. The network traffic is monitored in two ways: (i) the packets dropped by the iptables firewall are inspected (if any), and (ii) during the testing, `tcpdump` is used to capture the network traffic such that analysis can be performed later, if necessary. For step (2), it is possible to make the test results more robust by documenting all steps taken while testing the device to ensure reproducibility. However, producing precise but lengthy documentation on how to test the devices is out of scope for this research.

5.2.2 Criteria 2 and 3

Criteria 2 and 3 both revolve around the IoT device being the center of attention of an attacker. The most common way for an attacker to interact with its victims is by sending malicious traffic. Other methods, where the attackers do not pro-actively send traffic, also exist; for instance, registering an expired domain used by the devices and move from there. In order to verify whether criteria 2 and 3 are met, it is necessary to know more about the traffic the attackers are sending to the devices. There are multiple methods of investigating the attackers' traffic and its effects. Two of such methods are described below.

Replaying attack traffic

One of the methods is to replay attack traffic against a device while the profile is enforced. An advantage of this technique is that a simulation like this comes really close to reality because the attack is not just simulated but actually performed. As such, when the device is protected during this test, it is likely to be protected in the real world. A disadvantage is that, in order to replay attack traffic against a device, it is necessary to actually have network traffic of botnet activity. Such traffic is not always easy to get access to. Additionally, depending on the contents of the replayed traffic and on whether the profile actually succeeds in blocking the traffic, the attack against the device may actually succeed. For instance, if the replayed traffic would immediately add the device to

[1]Note that in this case, while the device *attempts* to contact a domain that is not in the profile, it will not actually *succeed* in doing so since the domain name is not in the whitelist. This event can be recognized from the pcap traces as follows: the device attempts to initiate a TCP connection (i.e., a TCP packet with the `SYN` flag set is sent), but the router replies with an ICMP Destination Unreachable message.

Abbreviation	Description	Involved protocol(s)
TCP/22	Dictionary attack against SSH daemon	TCP, port 22
TCP/23	Dictionary attack against telnet daemon	TCP, port 23 or 2323
TCP/80	Vulnerability in HTTP server	TCP, port 80 or 8000

Table 5.1: Summary of common methods to gain access to IoT devices used by botnets.

a botnet, the device could become a participant in the botnet. Therefore, when replaying attack traffic against a device, it is necessary to take appropriate precautions to prevent any of such problems.

Analyzing literature and attack traffic

Another method is to analyze literature and attack traffic (if available) and extract characteristics of the malicious network traffic. This can be done on different levels. For instance, it is possible to look at packet sizes or packet rates. Alternatively, it is possible to look at which transport layer protocols and port numbers were involved (for instance: telnet on TCP port 23). Furthermore, it is possible to perform deep packet inspection. In the case of the telnet protocol, the traffic is not encrypted. This makes it possible for an observer to determine how the attacker is interacting with the victim. For instance, it is possible to distinguish between failed login attempts and successful login attempts.

For this research, a theoretical approach towards gathering information about the attackers' activity is chosen. As such, literature has been studied and captured traffic will be analyzed for patterns. In particular, traffic will not be replayed against a device. We expect that a literature study and traffic analysis will provide a sufficient amount of information. It is uncertain how valuable replaying attack traffic would be while it would be time-consuming to create a proper setup in which malicious traffic can be replayed against devices in a safe manner.

We now have decided that we are going to use literature as a source of information regarding attackers' network. However, verifying whether criterion 2 holds needs different information than verifying whether criterion 3 holds. In order to verify whether criterion 2, *The enforced profile prevents the IoT device from being hacked*, is satisfied, it is necessary to know how attackers compromise devices and add the devices to their botnets. From the literature in Section 2.4, Table 5.1 has been derived which summarizes the common ways unauthorized access to IoT devices is gained. Those methods can be compared against the whitelist of incoming traffic in the generated profiles to see whether the malicious traffic would reach the device or not.

Conversely, in order to verify whether criterion 3, *The enforced profile prevents the IoT device from being misused in an attack if it were successfully hacked anyway*, is satisfied, the questions that need to be answered are (a) how do the attackers instruct the devices what to attack, and (b) what kind of traffic does the attack consists of. Answers to those questions can also be found in literature. We will use those answers to theorize whether a device would actually be vulnerable, for instance by comparing the generated profiles against the information found in the literature. From the literature reviewed in Section 2.4, we find that the answer to (a) is that the botnet is either controlled via a centralized infrastructure controlled directly by the attacker, or via a peer-to-peer network. In the former case, often HTTP or a custom protocol is used. From the same literature, we find that IoT botnets mostly carry out UDP and TCP floods. This answers (b).

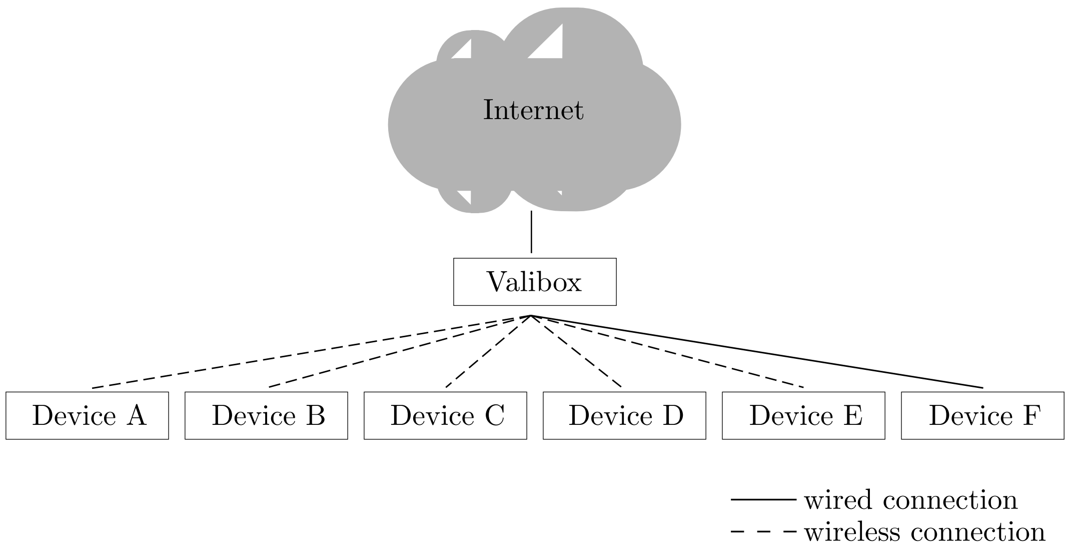


Figure 5.1: Schematic overview of the network that was used to evaluate the prototype. Legend: Device A: TP-Link LB100; Device B: VStarcam D1 Door Camera; Device C: Sonoff S20; Device D: Samsung TV; Device E: Motorola Moto X Force; Device F: Philips Hue Bridge.

ID	**Product name**	**Type of device**	**Network connection type**
F	Philips Hue Bridge	Light bulb controller	Wired
A	TP-Link LB100	Light bulb	Wireless
B	VStarcam D1 Door Camera	Doorbell with camera	Wireless
C	Sonoff S20	Power socket	Wireless
D	Samsung TV	TV	Wireless
E	Motorola Moto X Force[2]	Mobile phone	Wireless

Table 5.2: Devices that were used with the developed prototype.

5.3 Network Setup

In order to evaluate the prototype, it was necessary to deploy the prototype into a test network. This section describes the setup of the network that was used to evaluate the prototype.

The network closely resembled a typical home network as depicted in Figure 3.2. As is shown in Figure 5.1, the core of the network, consists of the Valibox mini-router (see Section 4.1). If possible, the devices were connected to the network using a wired connection. Those devices were connected to an Ethernet switch, and the Ethernet switch was connected to the LAN port of the Valibox. The other devices were connected to the network via the Wi-Fi access point provided by the Valibox. During the period in which network traffic was collected, the devices could access the Internet without any restrictions. Table 5.2 shows which devices were part of the network.

[2]While this phone is technically not an IoT device depending on which definition of IoT is used, verifying whether a profile can be generated and used successfully is still valuable.

Device	**Criterion 1 satisfied?**
Philips Hue Bridge	yes
TP-Link LB100	yes
VStarcam D1 Door Camera	yes
Sonoff S20	yes
Samsung TV	yes
Motorola Moto X Force	yes

Table 5.3: Shows per device whether criterion 1 is satisfied or not.

Host	**Protocol**	**Port number**	**Direction**
devices on local network	TCP	80	Incoming
`bridge.meethue.com`	TCP	80	Outgoing
`dcp.cpp.philips.com`	TCP	80	Outgoing
`dcs.cb.philips.com`	TCP	80	Outgoing
`diagnostics.meethue.com`	TCP	80	Outgoing
`fds.cpp.philips.com`	TCP	80	Outgoing
`time.meethue.com`	TCP	443	Outgoing
`ws.meethue.com`	TCP	443	Outgoing
`www.ecdinterface.philips.com`	TCP	80	Outgoing
`www2.meethue.com`	TCP	443	Outgoing
`0.openwrt.pool.ntp.org`	UDP	123	Outgoing
`1.openwrt.pool.ntp.org`	UDP	123	Outgoing
`2.openwrt.pool.ntp.org`	UDP	123	Outgoing
`3.openwrt.pool.ntp.org`	UDP	123	Outgoing
`time1.google.com`	UDP	123	Outgoing
`time2.google.com`	UDP	123	Outgoing
`time3.google.com`	UDP	123	Outgoing
`time4.google.com`	UDP	123	Outgoing
`valibox`	UDP	53	Outgoing
`valibox`	UDP	67	Outgoing

Table 5.4: Profile that was generated and enforced for the Philips Hue Bridge.

5.4 Evaluation Results and Discussion

The evaluation plan as outlined in Section 5.1 has been carried out on the network described in Section 5.3. The results are presented and discussed in this section.

5.4.1 Criterion 1

We first turn our attention to criterion 1, *The IoT device is able to function properly once a profile that has been generated for the device is enforced.* As is shown in Table 5.3, all tested features were observed to function correctly and monitoring the network traffic did not reveal any (attempts at) disallowed communication. As an example, Table 5.4 shows the profile that was generated and enforced for the Philips Hue Bridge. It shows that the device initiates contact to a number of HTTP/HTTPS services. Additionally, it uses the NTP and DNS protocols. Finally, it is shown that port 80 is being used by a device on the local network to contact the

Philips Hue Bridge. Using this port, a mobile phone is able to issue commands to the Philips Hue Bridge.

The Motorola Moto X Force is not an IoT device but a mobile phone. Therefore, we explicitly note how we evaluated the profile for this device. The usability test consisted of the following: it was verified that two messaging applications (Signal and WhatsApp) still function properly. Additionally, the Google Play Store was opened and the command was given to check for updates. Furthermore, the web browser was used to open a particular web site. We verified whether other web sites indeed were inaccessible, which was the case.

Discussion

As shown in Table 5.3 the devices subject to the profile enforcement were still able to carry out their functions correctly. However, a generated profile may become inaccurate due to changing network access patterns. This did not happen during the evaluation period but this can (and will) definitely happen in real-world usage. Network access patterns can change for a number of reasons:

Changes in user behavior
If the user starts using a feature that was not used during the data collection period, the device may attempt to access destinations that are not in the whitelist. This would result in a failure to contact that particular destination.

Software update
Additionally, when a software update is installed, it is possible that the new version of the software introduces new features that require connections to new destinations, for instance. We expect that devices that lean more towards specific-purpose computers (as opposed to general-purpose) will suffer less from these problems as the functionality of the device is more specific and as such will not change as much. Devices that lean more towards general-purpose computers are more likely to have the ability to install additional applications or have a web browser. For each additional application or web site, the profile may need to be adapted.

Fortunately, the developed prototype is capable of updating an existing profile. This means that if either of the above events happen, it is possible to update the profile to capture the new behavior. Note that this limitation only applies to our system that generates profiles automatically. Specifically, this is not a problem that has anything to do with the MUD specification. After all, if a device would support MUD, the profile would be built by the manufacturer and therefore, the profile should always be up to date.

A potential problem that remains is when devices, for some reason, alternate between domain names. For instance, the device could access a numbered resource. An example in this regard is the Samsung TV which was observed to connect to `otnprd8.samsungcloudsolution.net` and `otnprd11.samsungcloudsolution.net`. If, at a later point in time, the device would attempt to access `otnprd7.samsungcloudsolution.net`, a domain name that is not in the profile[3], accessing the service behind that domain name would fail. This in turn would render the generated profile inaccurate. In the case of the Samsung TV, the purpose of using domain names like this is uncertain; it is possible that it is used as a way to perform load balancing. Regardless of the precise reason, this poses a potential problem for automatically generated profiles. Note

[3]Due to time constraints, it was not verified whether this is actually the case.

that this is not a problem related to the MUD specification. After all, if manufacturers are responsible for providing an accurate profile, they can either list all domain names used for load balancing or choose to use another way of performing load balancing.

Furthermore, it is important to pay attention to a change of behavior that can occur in certain circumstances. For instance, while the Philips Hue Bridge was tested, it was verified what would happen when the device was not able to send any traffic to the Internet (except for DNS). It was observed that the device initially attempts to contact `bridge.meethue.com` and `www2.meethue.com`. When the device repeatedly does not succeed in connecting to those domains, it looks up `www.google.com`, `www.facebook.com`, `www.baidu.com` and `www.qq.com` and attempts to set up a HTTPS connection to these domains. This appears to be a connectivity check. This shows that a profile that is too strict can cause a device to change its behavior. It is important to pay attention to such behavior changes while working on generating and enforcing profiles for devices. Otherwise, the profile may contain unintended destinations.

We generated a profile for the Motorola Moto X Force, even though this mobile phone is not an IoT device. The device was not included because we expected to create a useful profile for it. Rather, it provided a good way to verify whether the profile was enforced properly. As a thought experiment, we will consider what a profile for a mobile phone would look like, even though that is not the main point of this thesis. Given the fact that the mobile phone is a general-purpose device, it is hard to create a narrow profile for the phone. The generated profile allows for using certain instant messaging applications, and those specific applications function correctly. However, for each application that is installed by the user, it is likely that the profile has to be updated. This is the case since a new application may contact destinations that are not in the whitelist. Furthermore, with an application such as a web browser, the list of hosts that may be contacted is infinite. Therefore, using a whitelist with domain names is not possible. In the case of a web browser, another approach may be more feasible. For instance, all outbound traffic to port 80 for HTTP and to port 443 for HTTPS could be allowed. While such a profile will allow the user to browse the Internet, it does not really restrict the network access of a device. As such, that profile may considered to be of limited value.

As stated in the previous paragraph, generating a useful MUD profile for a general-purpose device is hard or impossible. This is the case since the activities of general-purpose devices are very diverse by nature. Therefore, such activities are hard to capture in a narrow profile. However, we explicitly note that the results indicate that for specific-purpose devices such as IoT devices, it *is* possible to generate accurate MUD profiles. As Table 5.3 shows, enforcing those generated profiles does not impede the functionality of the device.

5.4.2 Criterion 2

We now turn our attention to criterion 2, *The enforced profile prevents the IoT device from being hacked.* The results are shown in Table 5.5. The table shows that a couple of devices have legitimate use for an open port, for instance the Philips Hue Bridge with TCP port 80 and the TP-Link LB100 with TCP port 9999. Furthermore, it shows that none of the devices need a telnet or SSH daemon that is accessible to the local network or to the Internet.

	From the literature			Found during evaluation
Device	**TCP/22**	**TCP/23**	**TCP/80**	
Philips Hue Bridge	no	no	yes	
TP-Link LB100	no	no	no	TCP/9999
VStarcam D1 Door Camera	no	no	no	
Sonoff S20	no	no	yes	
Samsung TV	no	no	no	
Motorola Moto X Force	no	no	no	

Table 5.5: Shows for each device whether a service was listening on a certain port and whether that service was observed to be used legitimately during the evaluation period. If that is the case, (possibly malicious) traffic on that particular port would be allowed by the profile that is generated for the device.

Discussion

Most attackers attack a device by sending traffic from the attacker towards the device. As all profiles deny connections from the Internet to be set up, all profiles pass this test. In particular, when a profile is enforced, it is not possible to connect to a telnet or SSH daemon, if it were present. To paint a more complete picture, we will now discuss what would happen if local traffic was also part of the profile. On the local network, legitimate cases have been observed where a local connection is made to a device. For example, the Philips Hue Android application initiates HTTP connections to the Philips Hue Bridge in order to control the light bulbs. If the server software listening on those ports would have a known vulnerability that can be used to gain access to the device, the generated profiles would not protect the devices from exploitation. However, it can be said that the attack surface is reduced; rather than leaving ports open of services that are not used by the user, only the ports necessary for using the device remain open.

For none of the examined devices, legitimate use of the telnet or SSH port was observed. We will now consider the security implications of the case where legitimate use of either protocols would have been observed anyway. In the case of legitimate use of telnet or SSH, the traffic could originate from two locations: the local network, or the Internet. Legitimate use originating from the Internet would have big security implications because it would be necessary to leave those ports accessible from the Internet. On the Internet, bots are continuously scanning telnet and SSH ports and attempting to break in. Use of strong passwords or authentication keys would help but that is not something the MUD specification can enforce. The other possibility is that telnet or SSH would be used from inside the local network. In this case, the threat of bots continuously scanning for open ports is not present. However, it is not uncommon for malware to attempt to spread to other devices in the local network of the compromised device. An example of such malware is Stuxnet [35]. As such, if another device in the network is compromised, a running telnet or SSH daemon poses a real risk as it could be used by such malware.

Finally, we will note that an attacker who uses a legitimate, whitelisted channel will not be stopped using such profiles. In other words, an attacker who interacts with the device through a whitelisted resource is free to attack the device. The generated profile will not impose any limitations onto traffic that flows through the legitimate channels. As such, while the use of a MUD profile will reduce the attack surface, it will not completely eliminate it.

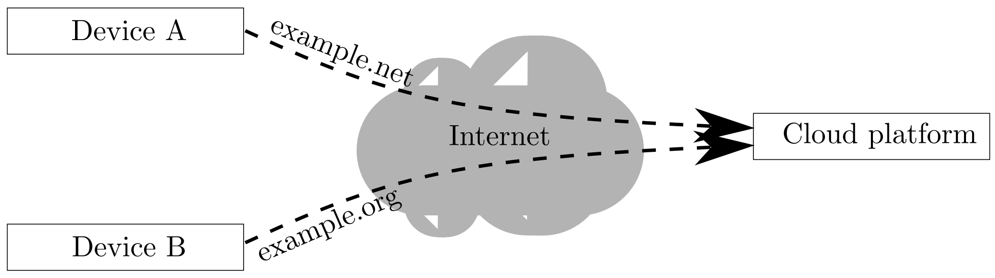


Figure 5.2: Device A and device B launch a denial of service attack against `example.net` and `example.org`, respectively. Both domains are hosted at the same cloud platform.

5.4.3 Criterion 3

Finally, we turn our attention to criterion 3, *The enforced profile prevents the IoT device from being misused in an attack if it were successfully hacked anyway.* The assessment of whether an IoT device can be used in a DDoS attack is highly dependent on the target that is chosen for the attack. After all, the whitelist in the profile lists specific domain names and IP addresses. Therefore, an attack launched by a botnet operator would be permitted by the profile if the target host and port are permitted in the profile.

Discussion

As was stated earlier, the majority of botnets support UDP and TCP floods to perform DDoS attacks. When a MUD profile is applied, such attacks can only be performed against services that are in the whitelist. As such, the number of possible victims is reduced from "everybody on the Internet" to "the destinations contained in the whitelist of the profile". This is an improvement, especially when considering that most DDoS targets will not be part of the whitelist. For instance, web sites of financial institutions or retailers - common targets of DDoS attacks [8] - are not present on the whitelist of a Philips Hue Bridge. Furthermore, in order for the attacker to be able to *command and control* the hacked device, the attacker should be able to interact with the device. Regarding peer-to-peer networks used for command and control, most peers will reside in residential networks. However, most profiles include specific services from vendors in their whitelists, not residential networks. As such, it is unlikely that a peer-to-peer botnet would be able to communicate effectively when the profile is enforced. If an attacker gains access to a server that is in the profile, the *infrastructure controlled by the attacker* could be used to interact with the devices. However, contrary to the situation when no profile is applied, the attacker must carefully choose its infrastructure rather than being able to use any server. This makes it harder for the attacker to successfully use infrastructure controlled by the attacker.

The use of cloud platforms is common nowadays. Examples of providers of such platforms are Amazon with Amazon Web Services and Google with Google Cloud Platform. A significant part of today's Internet runs on such cloud platforms [20]. This is also true for services used by IoT devices. For example, several of the investigated devices connect to cloud platforms of Samsung, Amazon and Google. Now imagine the following example scenario. Device A and device B are both IoT devices that reside somewhere in the Netherlands. Device A uses a service hosted at `example.net` while device B uses a service hosted at `example.org`. Both domains resolve to IP addresses that are hosted at the same cloud platform. The cloud platform provider uses anycast, such that those IP addresses are announced from multiple locations around the world. In this example, we assume that as a result, any packet originating from a host residing in the Netherlands to those IP addresses will be served by their data center in Amsterdam (users in other countries are served by other data centers). Now imagine that device A starts a DDoS attack against `example.net` while device B launches an attack against `example.org` (see Figure 5.2). Since `example.net` and `example.org` are both hosted at the same cloud platform, effectively they are attacking the infrastructure. If the DDoS attack succeeds, the services provided by that cloud platform would be unavailable to clients in the Netherlands. This despite the fact that according to the profiles, they are attacking different services. Successfully attacking a big cloud platform can impact many clients. With this problem in mind, one could argue that while technically the number of destinations that can be attacked is low, the number does not have to be very high since a significant part of today's Internet is hosted by only a couple of companies anyway[4]. One way to mitigate this problem is to apply rate limiting to IoT devices. This would make sure that all devices would still be able to access the services they need in order to operate. At the same time, the usefulness of the devices in a DDoS attack would be reduced. Currently, the MUD specification does not include rate limiting. As such, rate limiting could be an interesting addition or extension to the specification.

5.4.4 Summary

Summarized, enforcing the generated profiles allow the devices to function correctly (criterion 1). However, it is expected that profiles may become inaccurate due to changing user behavior or software updates. Additionally, enforcing the generated profile reduces the attack surface but a risk still remains (criterion 2). Furthermore, when a device is hacked anyway, the enforced profile will reduce the amount of damage an attacker can do, with some caveats (criterion 3). It is probably a good idea to extend MUD such that it becomes possible to apply rate limiting to an IoT device.

5.5 Prototype Limitations

The implemented prototype has a number of limitations (in addition to the limitations outlined in Section 4.6.1). Some of these limitations result from a shortcut being taken during the development of this prototype. Other limitations result from a changing Internet landscape. The limitations can be addressed by spending more time on the development of the software.

The prototype looks up domain names embedded in the profiles upon profile enforcement (see Section 4.6). A better method would be to do this when the device actually requests the domain name (see Section 3.3). During the evaluation period, this shortcut did not cause any problems. However, the longer a profile is enforced, the bigger the chance that the cached IP address for a domain name becomes invalid and the profile does not work as well anymore. This is not a fundamental problem with the chosen approach; it only is a shortcut taken during the development of the prototype which could be addressed when more time is available.

Furthermore, it is the case that the prototype does not actually filter any DNS responses. As a consequence, when a device looks up a domain that it should not be able to access, it will still receive the resulting DNS answer. With this implementation, it is possible for the device

[4]It should be noted that those big service providers are probably better at handling DDoS attacks than small service providers.

to look up a domain (`notallowed.example.net`) that it is not allowed to access. However, it is possible that a domain that the device is allowed to access (`allowed.example.org`) resolves to the same IP address as `notallowed.example.net`. This can happen with shared hosting or when using cloud services. Then the device may be able to access a service it should not be allowed to access. Additionally, not filtering DNS responses allows DNS to be used as a command and control mechanism [54].

Another limitation is that the prototype currently does not work with encrypted types of DNS such as DNS over TLS and DNS over HTTPS. As those methods become more common, the current method of annotating IP addresses in flow records with domain names may cease to work. The problem could be addressed by forcing IoT devices to use the resolver on the local network. The local resolver could then be instrumented to share information with the software developed for this research.

Chapter 6

Conclusion

6.1 Conclusion

At the beginning of this thesis, the main question of this research was defined:

> To what extent can automatically generated MUD profiles be used to prevent IoT devices from being hacked and/or from being misused in DDoS attacks?

To answer the main question, four Research Questions were defined which should aid answering the main question. The results of RQ1, *What information is needed to generate a MUD profile of an IoT device?*, were used to design and implement a system that can generate profiles of network access. Since the result of RQ1 does not directly contribute to answering the main question (but only indirectly by assisting in designing the system), the results of RQ1 are not repeated here. RQ2, RQ3 and RQ4 were answered in Chapter 5. Those Research Questions were answered by translating them into criteria. For each of the criteria, it was verified whether the prototype satisfies the criterion. This was done by performing both practical and theoretical analysis.

The answer to RQ2, *Are IoT devices able to function properly once generated MUD profiles are enforced?*, is that profiles can be successfully generated and enforced for the devices used during the evaluation. A foreseen problem is that legitimate network access patterns may change and as a result, the profile may become inaccurate. However, the implemented prototype is able to update an existing profile which may remedy that problem. More work in this area is necessary.

Additionally, to answer RQ3, *Does enforcing the generated MUD profile prevent IoT devices from being hacked?*, it was found that an enforced profile reduces the attack surface of the devices. Therefore, it becomes harder to gain unauthorized access to the devices. As such, enforcing the profile will reduce the chance of the device being hacked. Nevertheless, some risk still remains. In particular, it is the case that enforcing such profiles will not help against hacking attempts that occur using the whitelisted paths.

Furthermore, to answer RQ4, *If an IoT device were hacked anyway, does enforcing a MUD profile prevent IoT devices from being misused in (for instance) a DDoS attack?*, it was found that enforcing such profiles reduces the amount of damage a hacked device can do. In particular, in order to be able to attack a certain victim, the victim needs to be in the whitelist of the profile. Especially in the case of specific-purpose devices with strict profiles, this reduces the number of destinations a device can legitimately (or illegitimately) contact. However, given the fact that infrastructure is heavily shared these days (e.g. through the use of cloud platforms), it will be necessary to apply rate limiting as well, for instance.

Finally, to answer the main question, it is indeed possible to successfully generate profiles that can be applied to devices. It reduces the chances of being hacked and being misused in DDoS attacks. However, to further reduce those chances, additional measures such as rate limiting may be necessary. Another important finding of this work is that the evaluation results indicate that profiles can be generated automatically from traffic traces for specific-purpose devices without impeding the functionality of the device. This is considered to be hard or even impossible for general-purpose devices.

6.2 Future Work

The work presented in this thesis can be improved in a number of ways. As mentioned earlier, the usefulness of rate limiting should definitely be investigated. Additionally, the prototype can be improved to eliminate the shortcuts and limitations as described in Section 4.6.1 and Section 5.5. Furthermore, it would be interesting to know how well updating a generated profile works.

As shown by this research, IoT devices contain server software which listens on certain ports. It would be interesting to look deeper into the implementation of this software. For instance, the server software may contain undiscovered vulnerabilities or other implementation errors. Additionally, it would be interesting to investigate which cloud platforms are used most frequently by services used by IoT devices.

Finally, once IoT devices that support MUD are available on the market, it would be interesting to investigate whether the profiles provided by the manufacturers are as narrow as they could be. In other words, do the profiles provided by the manufacturers only allow traffic that is absolutely necessary to use the device or could the profile be even more strict? This could be done by comparing the profile provided by the manufacturer with a profile that is generated using the approach described in this thesis.

Appendix A

Implementation Considerations

A.1 Software

Figure 4.3 shows a number of software components. This section will go into the choices that had to be made before the first line of code could be written, such as which programming languages and libraries to use.

Most of the software described in this overview is intended to run on the home router. In fact, the only program which does not necessarily have to run on the home router is the traffic collector program. For all the other programs, it makes sense that they run on the home router. For traffic collection and profile enforcement, this is an absolute requirement. For profile generation, the benefit of running the code on the router is that the router will be able to operate independent from any other device in the network.

For the programs that need to run on the home router, it was chosen to use the Lua program language. One good reason for chosing to do so is the fact that a part of the SPIN code base is already written in Lua. Therefore, there is no need to install additional runtime environments or to drastically change the build process of the Valibox OpenWRT image. The author did not make the decision to use Lua, but certainly can give a couple of reasons why Lua is an appropriate choice. For instance, Lua is a lightweight programming language; it does not need a heavy runtime environment to run. This is an advantage in a resource-constrained environment like a router. Additionally, compared to a programming language such as C, with Lua it is harder to shoot oneself in the foot with certain types of bugs, such as buffer overflows.

In the case of the traffic collection program, it would be nice if the program can run on the home router but it is by no means necessary; it is also possible to process a pcap on a desktop computer and upload the result to the home router (alternatively, the output could be published onto the message bus). As such, in order to make it possible to run the program on the Valibox, the initial language choice for this program would be Lua. However, no suitable bindings to the pcap library were found. As such, it was chosen to use the C programming language. The program only depends on `libpcap` and `ldns` so in terms of dependencies, the program should be lightweight enough to run it on the Valibox.

A.2 Using the Prototype

To show how the different components of the prototype work together, this section shows the sequence of actions that should be performed in order to enforce a generated profile for a device. It shows that the system can actually be used in practice. Figure 4.3 shows how the different components interact. This section explains how those different components are named. Additionally, it explains how to use the software step-by-step.

1. The first step consists of collecting information about the network activity of the device. The *Database writer* is implemented as the `mqtt_nm.lua` program. To start this program, it is sufficient to ssh to the Valibox and execute the `mqtt_nm.lua` program.

   When the network activity will be collected using the SPIN software, no additional steps are necessary since the SPIN software is started automatically at startup.

   When a pcap file with previously collected network traffic is available, the `pcap-spin-json` program can be used instead of the SPIN software. The program can be used on a UNIX workstation. To publish the results to the `SPIN/traffic` MQTT channel, the `mosquitto_pub` program is used. The `run.sh` shell script is a convenient wrapper around `pcap-spin-json` and `mosquitto_pub` that takes two arguments: the pcap file to be read and the IP address of the device running the MQTT message broker (the Valibox).

2. The next step is to connect the device to the network and turn it on. It is important to start the measurements *before* turning on the device in order to create a complete picture of what the device is doing.

3. Use the device. During this time, it is necessary to exercise all features of the device in order to trigger all possible network traffic that can reasonably be expected to occur. Furthermore, the measurement should be over a timespan that is long enough. This is a good way to make sure periodic things, such as an update check, is observed and recorded.

4. Stop the measurement. This is done by stopping the `mqtt_nm.lua` program that was started during step 1.

5. Generate the profile. This is done by using the `generate-profile.lua` program on the Valibox. Passing the `-a` flag will generate profiles for all devices found on the network. Pass the `-j` flag to export the profile in JSON format and redirect the output to a file.

6. Enforce the generated profile. This is done by invoking the `generate-fw.lua` program twice. Initially, the `-i` flag must be passed to the program to generate rules that initialize the firewall. Afterwards, the program should be invoked with `-e` and the profile generated in step 5 should be provided on `stdin`.

The source code of the prototype can be found at `https://schutijser.com/msc-thesis-code/`.

Bibliography

[1] About - Civil Infrastructure Platform. `https://www.cip-project.org/about`. Accessed Mar 29, 2018.

[2] IoT_reaper: A Few Updates. `https://blog.netlab.360.com/iot_reaper-a-few-updates-en/`. Accessed Apr 3, 2018.

[3] IoT_reaper: A Rappid Spreading New IoT Botnet. `https://blog.netlab.360.com/iot_reaper-a-rappid-spreading-new-iot-botnet-en/`. Accessed Apr 3, 2018.

[4] Members - Civil Infrastructure Platform. `https://www.cip-project.org/members`. Accessed Mar 29, 2018.

[5] IEEE Standard for Local and metropolitan area networks - Secure Device Identity. *IEEE Std 802.1AR-2009*, pages 1–77, Dec 2009.

[6] Dyn Analysis Summary Of Friday October 21 Attack. `https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/`, 2016. Accessed Mar 28, 2018.

[7] 360 Netlab. Botnets never Die, Satori REFUSES to Fade Away. `https://blog.netlab.360.com/botnets-never-die-satori-refuses-to-fade-away-en/`. Accessed Jul 10, 2018.

[8] Akamai. State of the Internet - Q4 2017. `https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/q4-2017-state-of-the-internet-security-infographic.pdf`, 2017. Accessed Aug 14, 2018.

[9] M. Anagnostopoulos, G. Kambourakis, P. Kopanos, G. Louloudakis, and S. Gritzalis. Dns amplification attack revisited. *Computers & Security*, 39:475 – 485, 2013.

[10] K. Angrishi. Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets. *arXiv preprint arXiv:1702.03681*, 2017.

[11] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the Mirai Botnet. In *USENIX Security Symposium*, 2017.

[12] B. N. Astuto, M. Mendonça, X. N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *Communications Surveys and Tutorials, IEEE Communications Society*, 16(3):1617 – 1634, 2014. accepted in IEEE Communications Surveys & Tutorials.

[13] I. N. Bermudez, M. Mellia, M. M. Munafo, R. Keralapura, and A. Nucci. DNS to the Rescue: Discerning Content and Services in a Tangled Web. In *Proceedings of the 2012*

Internet Measurement Conference, IMC '12, pages 413–426, New York, NY, USA, 2012. ACM.

[14] N. Brownlee, C. Mills, and G. Ruth. Traffic flow measurement: Architecture, October 1999. RFC2722.

[15] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[16] B. Claise. Cisco Systems NetFlow Services Export Version 9, October 2004. RFC3954.

[17] J. Corbet. Super long-term kernel support. `https://lwn.net/Articles/749530/`, 2018. Accessed Mar 29, 2018.

[18] A. Cunningham. Why isn't your old phone getting Nougat? There's blame enough to go around. `https://arstechnica.com/gadgets/2016/08/why-isnt-your-old-phone-getting-nougat-theres-blame-enough-to-go-around/`. Accessed Mar 29, 2018.

[19] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 435–448, New York, NY, USA, 2014. ACM.

[20] Elizabeth Weise. Does Amazon control the Internet, or does it just feel that way? `https://eu.usatoday.com/story/tech/talkingtech/2017/03/01/amazon-control-internet-aws-cloud-services-outage/98548762/`, 2017. Accessed Aug 23, 2018.

[21] M. Feily, A. Shahrestani, and S. Ramadass. A Survey of Botnet and Botnet Detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273, June 2009.

[22] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[23] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing, May 2000. BCP38.

[24] GL-iNet. GL-iNet GL-AR150 Mini Router User Guide. `https://static.gl-inet.com/www/uploads/2017/06/minirouter_user-guide_AR150_20170629.pdf`. Accessed Jun 29, 2018.

[25] M. G. Gouda and A. X. Liu. A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137, June 2005.

[26] G. M. Graff. How a Dorm Room Minecraft Scam Brought Down the Internet. `https://www.wired.com/story/mirai-botnet-minecraft-scam-brought-down-the-internet/`, 2017. Accessed Feb 27, 2018.

[27] J. Habibi, D. Midi, A. Mudgerikar, and E. Bertino. Heimdall: Mitigating the Internet of Insecure Things. *IEEE Internet of Things Journal*, 4(4):968–978, 2017.

[28] A. Hamza, D. Ranathunga, H. H. Gharakheili, M. Roughan, and V. Sivaraman. Clear as MUD: Generating, Validating and Applying IoT Behaviorial Profiles (Technical Report). *arXiv preprint arXiv:1804.04358*, 2018.

[29] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita. Botnet in DDoS Attacks: Trends and Challenges. *IEEE Communications Surveys Tutorials*, 17(4):2242–2270, Fourthquarter 2015.

[30] IEEE. IEEE Std 802.1AR-2018 (Revision of IEEE Std 802.1AR-2009) - IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity. `https://standards.ieee.org/findstds/standard/802.1AR-2018.html`. Accessed Aug 1, 2018.

[31] IETF. Operations and Management Area Working Group (opsawg). `https://datatracker.ietf.org/wg/opsawg/about/`. Accessed Mar 15, 2018.

[32] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE communications surveys & tutorials*, 16(2):898–924, 2014.

[33] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[34] B. Krebs. Reaper: Calm Before the IoT Security Storm? `https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/`, 2017. Accessed Mar 24, 2018.

[35] R. Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security Privacy*, 9(3):49–51, May 2011.

[36] E. Lear. Re: [OPSAWG] draft-ietf-opsawg-mud. `https://www.ietf.org/mail-archive/web/opsawg/current/msg05441.html`, 2018. Accessed Feb 26, 2018.

[37] E. Lear, R. Droms, and D. Romascanu. Manufacturer Usage Description Specification. `https://datatracker.ietf.org/doc/draft-ietf-opsawg-mud/18/`, 2018. Accessed Mar 16, 2018.

[38] S. A. Mehdi, J. Khalid, and S. A. Khayam. Revisiting traffic anomaly detection using software defined networking. In *International workshop on recent advances in intrusion detection*, pages 161–180. Springer, 2011.

[39] R. Mortier, T. Rodden, T. Lodge, D. McAuley, C. Rotsos, A. W. Moore, A. Koliousis, and J. Sventek. Control and understanding: Owning your home network. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–10. IEEE, 2012.

[40] National Institute of Standards and Technology. Cve-2018-10088. `https://nvd.nist.gov/vuln/detail/CVE-2018-10088`, 2018.

[41] Y. M. P. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. IoTPOT: analysing the rise of IoT compromises. *EMU*, 9:1, 2015.

[42] A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks*, 51(12):3448–3470, 2007.

[43] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), Apr. 2007.

[44] J. Postel. User datagram protocol, August 1980. RFC0768.

[45] J. Postel. Transmission control protocol, September 1981. RFC0793.

[46] N. Provos et al. A Virtual Honeypot Framework. In *USENIX Security Symposium*, volume 173, pages 1–14, 2004.

[47] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *Systems and Networks Communications, 2008. ICSNC'08. 3rd International Conference on*, pages 23–26. IEEE, 2008.

[48] B. Schneier. The Internet of Things Is Wildly Insecure - And Often Unpatchable. `https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html`, 2014. Accessed Mar 16, 2018.

[49] B. Schneier. Your WiFi-Connected Thermostat Can Take Down the Whole Internet. We Need New Regulations. `https://www.schneier.com/essays/archives/2016/11/your_wifi-connected_.html`, 2016. Accessed Aug 12, 2018.

[50] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 213–226, New York, NY, USA, 2015. ACM.

[51] SIDN. SIDN website. `https://www.sidn.nl/`. Accessed Aug 1, 2018.

[52] SPIN Contributors. SPIN Core Software. `https://github.com/SIDN/spin/blob/master/README.md`. Accessed Aug 1, 2018.

[53] The PostgreSQL Global Development Group. Network address types. `https://www.postgresql.org/docs/10/static/datatype-net-types.html`. Accessed Jul 3, 2018.

[54] M. v. Steen, C. Rossow, N. Pohlmann, H. Bos, F. C. Freiling, and C. J. Dietrich. On botnets that use dns for command and control. In *2011 Seventh European Conference on Computer Network Defense(EC2ND)*, volume 00, pages 9–16, 09 2011.

[55] S. T. Zargar, J. Joshi, and D. Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.