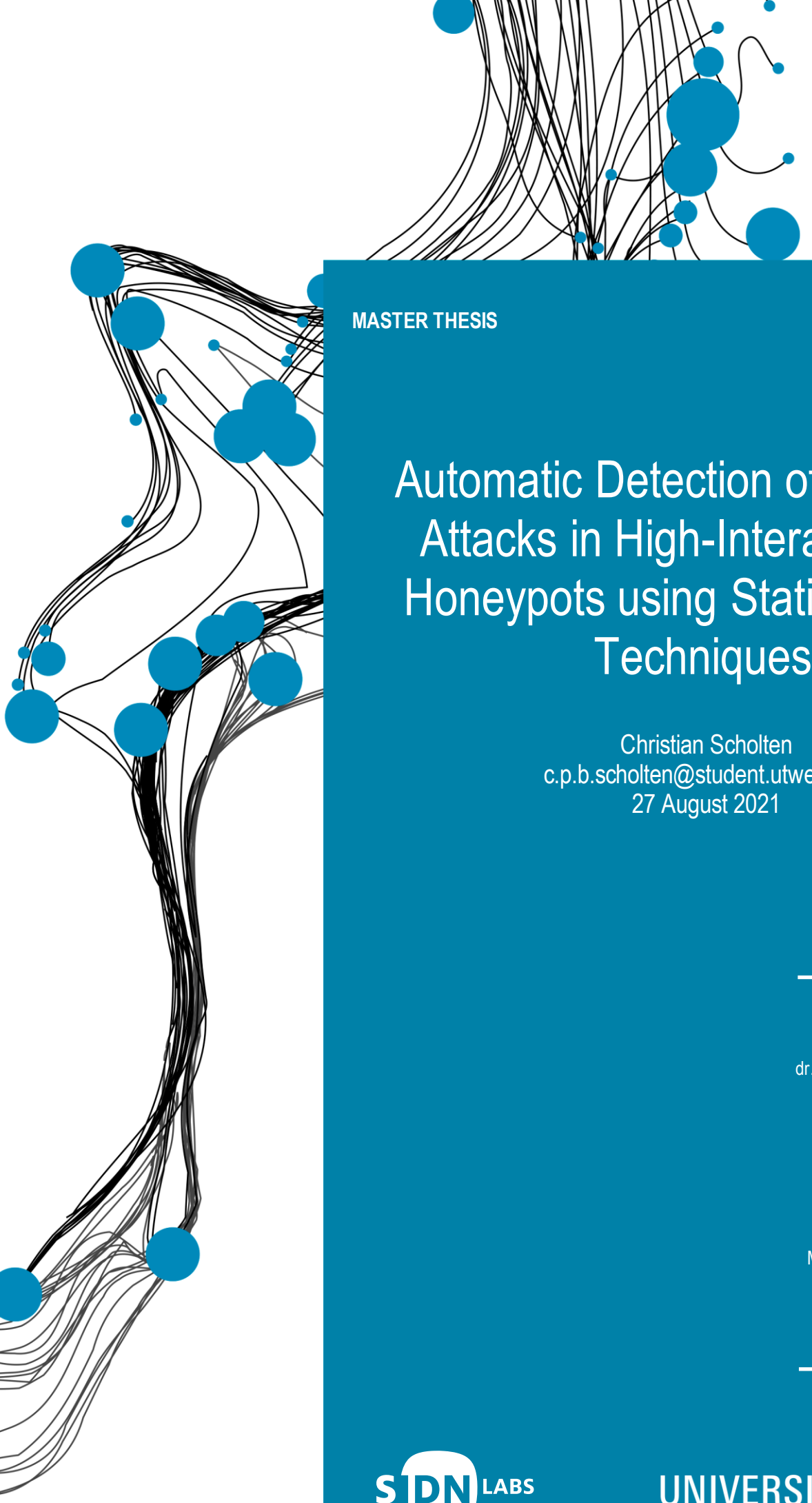

MASTER THESIS

Automatic Detection of Zero-Day Attacks in High-Interaction IoT Honeypots using Static Analysis Techniques

Christian Scholten
c.p.b.scholten@student.utwente.nl
27 August 2021

SUPERVISION
prof. dr. ir. R. M. van Rijswijk-Deij
dr. ir. A. Continella
dr. ir. E. E. H. Lastdrager (SIDN Labs)

EXAMINATION COMMITTEE
prof. dr. ir. R. M. van Rijkwijk-Deij
dr. ir. A. Continella
prof. dr. ir. A. Pras

DACS and SCS
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

SIDN LABS

UNIVERSITY OF TWENTE.

Samenvatting

Onveilige Internet of Things (IoT) apparaten zijn een populair aanvalsdoel worden gebruikt in botnets om Distributed Denial of Service-aanvallen (DDoS) uit te voeren. Om IoT apparaten beter te beschermen is het belangrijk om te weten welke aanvalsvectoren gebruikt worden om deze apapraten aan te vallen en welke kwetsbaarheden misbruikt worden. Honeypots kunnen gebruikt worden om deze vectoren te vinden. De meeste honeypots maken gebruik van IDS-signatures van al bekende kwetsbaarheden. Deze signatures zijn niet geschikt om nieuwe aanvalsvectoren te vinden waarvoor geen signature bestaat.

In deze scriptie onderzoeken we een nieuw soort honeypot voor IoT-firmware die statische analyse gebruikt om automatisch kwetsbaarheden in firmware images te vinden. Met firmware re-hosting tools kan de firmware geëmuleerd worden en als honeypot gebruikt worden. Op deze honeypot kunnen debugging tools, zoals GDB, gebruikt worden om aanvallen op de nieuwe kwetsbaarheden te detecteren. Ook kan de honeypot gebruikt worden om debugging rules toe te voegen. Deze rules specificeren de eigenschappen van een kwetsbaarheid input en kan verifieren of een verzamelde input met de honeypot deze kwetsbaarheid misbruikt om aanvallen te detecteren waarvoor geen signature bestaat.

Aanvallen kunnen met hoge accuraatheid gedetecteerd worden in de honeypot. De evaluatie van de honeypotdata en van proof-of-concept aanvallen liet zien dat de debugging rules een goed alternatief is om accuraat nieuwe aanvallen te detecteren waarvoor geen signatures bestaan. Het detecteren van aanvallen met statische analyse, daarentegen, is gelimiteerd door het kleine aantal images dat geëmuleerd kan worden met firmware re-hosting en waarin de statische analyse een kwetsbaarheid kan vinden. Vanwege deze limitatie heeft het gebruik van statische analyse weinig voordeel ten opzichte van de debugging rules en IDS signatures. In het toekomstige werk kunnen de debugging rules uitgebreider geëvalueerd worden om een overzicht te krijgen van de toegevoegde waarde van deze rules op een groter aantal firmware images.

Abstract

Insecure Internet of Things (IoT) devices are popular targets for use in botnets or for launching Distributed Denial of Service (DDoS) attacks. In order to secure these devices, it is vital to gain a proper understanding of the current attacks targeting these devices and to discover which vulnerabilities are actively being targeted. Honeypots can provide insight the current landscape of attack targeting IoT devices. The detection mechanisms of honeypots rely in most cases on IDS-signatures of already known attacks, which are not suitable for discovering novel attack types for which no signature exists.

In this research, we investigate a new type of honeypot for IoT firmware images that combines static analysis and firmware re-hosting tools to automatically detect vulnerabilities in firmware. The firmware re-hosting tool is used to host a honeypot in combination with debugging tools, such as GDB, to detect attacks targeting the vulnerabilities. Secondly, the honeypot allows to add debugging rules. These rules specify a vulnerable input source, verify that a vulnerability can be triggered and detect attacks targeting vulnerabilities for which no signature exists.

Our solution is able to accurately detect vulnerabilities using either the static analysis or the manually added debugging rules. The evaluation of the honeypot data and proof-of-concept attacks showed that the manually added debugging rules are a useful solution for accurately detecting attacks novel attacks. Deployment of the system proved to be limited by the small subset of firmware images that can be emulated using firmware re-hosting tools and firmware images in which static analysis was able to discover vulnerabilities. Due to this limitation, the use of static analysis techniques for emulated IoT firmware does not provide sufficient benefits. Future work can be done to perform a more elaborate investigation of the custom debugging rules.

Acknowledgements

I would like to thank my supervisors, Elmer Lastdrager from SIDN Labs, Roland van Rijswijk-Deij from the DACS group of the University of Twente and Andrea Continella from the SCS group of the University of Twente for their guidance, feedback, support and ideas. I would also like to thank Aiko Pras for joining the graduation committee as a senior examiner. Secondly, I would like to thank my colleagues at SIDN Labs. Even though the majority of the time was spent from home due to COVID restrictions, I still enjoyed my time there.

Furthermore, I would like to thank Elmer Lastdrager, Roland van Rijswijk-Deij, Andrea Continella and my brother, Stefan Scholten, for reading earlier versions of my thesis. Their feedback was valuable to me.

This project would not have been possible without the extensive work done prior by the developers of Karonte and FirmAE. I would like to explicitly thank them for making this research possible.

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

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet has become more interconnected than ever before. According to estimates by Gartner, there are an approximated 5.8 billion enterprise Internet of Things (IoT) endpoints in use at the end of 2020, an increase of 21% compared to 2019 [1]. This influx of new devices creates new challenges in securing the Internet as a whole, because an estimated 57% of these devices are vulnerable to at least one vulnerability with a medium or high risk level on the CVSS scale [2]. The CVSS scale is a standard used to create a qualitative representation of the severity of a vulnerability [3].

Many IoT devices are set up with default passwords and have limited support due to the low cost and hardware limitations. This makes IoT devices a common target for attackers [4]. Furthermore, attacks can easily be scaled to target thousands of targets by leveraging search engines for Internet-connected devices, such as Shodan [5], and network scanning tools, such as ZMap [6], to discover devices vulnerable to a certain exploit. Securing these IoT devices is essential to protect the Internet against large scale botnets, like Mirai [7], targeting these devices for large Distributes Denial-of-Service (DDoS) attacks. To achieve this, we need novel ways of detecting vulnerabilities in devices.

New developments in firmware re-hosting have shown an increasing capability of emulating device firmware outside of the original device. A recent example of this is FirmAE [8]. These tools have the advantage that they allow deeper introspection into the system, with full access to the system. This allows us to employ static analysis techniques that allow for far richer information on intrusions than traditional analysis of network traffic and system logs.

The goal of this research is to develop a system that can detect novel attacks attacks based on High-Interaction IoT honeypots in combination with static analysis techniques to automatically detect attacks targeting IoT devices.

Static analysis tool such as Karonte [9] can be used to discover vulnerable execution paths in device firmware. The honeypots are based on firmware re-hosting to allow for scalability without the need to own the physical device. The detection of zero-day attacks is performed by collecting the user inputs from a high-interaction IoT honeypot. A vulnerable execution path is then reconstructed using the information

from the static analysis to discover if the vulnerability can be reached and triggered using the inputs collected by the honeypot. The generated alerts can then be evaluated in terms of completeness and soundness. Finally, the performance of the system can be evaluated to discover if static analysis provides a tangible benefit over an IDS-based solution.

1.1 Research Questions

This project is done in context of the Security and Privacy for In-home Networks (SPIN) platform [10, 11]. To accurately protect the users of this platform, it is vital to discover which attacks are targeting their devices and to protect the users from further intrusions. Current high-interaction IoT honeypots usually detect malicious traffic using an IDS-based solution [12, 13, 14]. These solutions are dependent on signatures and may miss zero-day attacks. Honeypots with dynamic analysis techniques, on the other hand, require a large computational overhead [15].

Static analysis techniques may be a possible solution to reducing the computational overhead, while still being able to detect zero-day attacks. Static analysis tools such as Karonte [9] can be used to model the device firmware and each possible insecure interaction. User inputs on honeypot can then be collected and used to reconstruct a symbolic execution state to discover whether these inputs can reach and trigger a vulnerability.

This results in the following main research question:

To what extent are static analysis techniques effective in automating the detection of zero-day attacks in high-interaction IoT honeypots?

To answer this main question, we break this main question down into the following concrete research questions:

RQ1. *How are the input sources of an IoT honeypot chosen to drive the static analysis?*

Not every vulnerability that is identified through static analysis is exploitable through inputs. This leads us to the next research question:

RQ2. *What steps need to be taken to determine if an input can reach and trigger a vulnerability?*

To evaluate the system, we need to consider the performance of the system to discover if the system can accurately detect attacks. This results in the third research question:

RQ3. *How accurate are the alerts triggered by the static analysis using performance metrics?*

Another aspect we need to consider is if our solution provides a tangible benefit over existing solutions. In the final research question, we compare it against IDS-based solutions:

RQ4. *How does the performance of the static analysis compare to an IDS-based approach?*

1.2 Structure

The report starts with the background and related work in Chapter 2. In this chapter, we will discuss the current landscape of IoT Security, the *state-of-the-art* in IoT honeypots, firmware re-hosting tools and automatic vulnerability detection techniques.

Chapter 3 will discuss the approach used to create and evaluate the prototype. After the approach, we will discuss the individual components of the prototype and the methodology used to create each of the components in Chapter 4. This prototype is then deployed as a honeypot in order to gather inputs for the evaluation. The details regarding the deployment are discussed in Chapter 5.

Finally, using the honeypot data and proof-of-concept attacks, we will evaluate the prototype in Chapter 6. Chapter 7 concludes this research and will answer the main question.

Chapter 2

Background and Related Work

The related work and background related to this research is introduced in this chapter. This will be structured as follows: Section 2.1 evaluates the current landscape of IoT devices and the main security concerns of these devices. Section 2.2 evaluates the current developments in honeypots and re-hosting of firmware images. Section 2.3 concludes this chapter, which describes the state-of-the-art in the automated detection of vulnerabilities and zero-day attacks.

2.1 IoT security landscape

With the large increase in active IoT devices, it is important to discuss the current state of security of IoT devices, the threats that are targeting IoT devices and legal developments in IoT security.

Many IoT devices have poor security standards. Morgner and Benenson [16] used the ZigBee protocol as an example to argue that a *market-for-lemons* applies to security of IoT devices. This economic theory states that users are unwilling to pay for something they cannot assess. This asymmetric information of the security of such devices at the consumer level creates little incentive for manufacturers to invest in security and privacy. Instead, most manufacturers prioritize a rush to market to gain a larger foothold.

This combination of large scale-deployment of IoT devices and poor security creates a perfect target for attacks. Antonakakis et al. [7] showed that many IoT devices were usually configured with default passwords and many ports not necessary for the functioning of the device left open. This allowed the Mirai botnet to spread rapidly and launch large DDoS attacks with simple brute-force attacks on Telnet services with a small dictionary of usernames and passwords.

The Unit42 2020 IoT Threat Report [2] by Palo Alto Networks estimates that currently 98% of enterprise IoT device traffic is unencrypted, which may expose personal and confidential traffic. Furthermore, encryption, when implemented, is often sub-standard. The LoRaWan protocol uses AES-CTR with the packet counter as a nonce. This allows for a *crib-dragging* attack [17]. With the encryption key retrieved from such an attack,

an adversary can inject fake data. A whitepaper by IOActive gives the example of an attack on a LoRaWAN pressure sensor monitoring a tank containing flammable goods. Injecting fake data indicating a low pressure could be used to raise the temperature and cause a pipe to break or explode [18].

Updating IoT devices is not always easy or possible. LoRaWan devices only have limited bandwidth, which makes updating these devices over the network infeasible in most cases. Another obstacle is that many chipsets driving IoT devices are using closed-source drivers called *binary blobs*. These are usually supported only for a certain period of time and only work with a single software version, e.g. Linux Kernel 5.10. If this version goes end-of-life, then it becomes difficult to continue support for the device. Smartphone manufacturers suffer from similar problems, although Qualcomm recently pledged to extend security support to four years for their upcoming Snapdragon processors [19].

There have been proposals in government to regulate the security of IoT devices. A recent development is a new European standard to harmonize security practices for IoT manufacturers [20]. Such a standard will help to ensure minimum security and update requirements for future IoT devices. Other developments include the California IoT law [21] that requires "any device, or other physical object that is capable of connecting to the Internet, directly or indirectly, and that is assigned an Internet Protocol address or Bluetooth address" to equip a "reasonable security feature". Still, this law allows for a very broad definition of security features.

This section has shown that there are still lots of improvements to be made to improve the security of IoT devices. Therefore it is important to gain insight in the current attacks targeting these devices. Honeypots are one of the ways that can be used to gain an overview of the current attacks targeting these devices, which will be discussed in the next section.

2.2 Honeypots

A honeypot, as defined by Fan et al. [22], is an information system with the aim of using information resources to attract unauthorized and illicit access with the purpose of security investigation. Their definition includes two essential elements, decoys and captor. The decoy can be any kind of information system, while the captor facilitates all security-related functions. Mokube and Adams [23] define three main purposes of a honeypot: to distract users from more valuable machines on a network, to provide early warnings about new attacks and exploits and to allow in-depth analysis of exploits. In this section we will discuss the recent developments in honeypots, the use of firmware re-hosting for honeypots and the ability to fingerprint of honeypots.

2.2.1 Levels of interaction

There are several levels of interaction that a honeypot can provide. We will discuss the state-of-the-art for each of these levels of interaction.

Low-interaction honeypots A low-interaction honeypot only provides an attacker with a predefined number of interactions and is mostly a static environment. The classical example of this is honeyd [24], a framework that can be used to create multiple virtual hosts that appear to be running arbitrary services. Another example is the honeypot in the Ubiquiti UDM router that passively detects LAN devices attempting to gain access to unauthorized services [25].

Luo et al. [26] have recently introduced IoTCandyJar. This paper takes a unique spin on low-interaction honeypots and uses machine learning to learn correct responses for requests to make a low-interaction honeypot appear more genuine. Such an approach allows for scalability and automated deployment of many different honeypots emulating different types of devices.

Medium-interaction honeypots A medium interaction honeypot emulates parts of a system and allows an adversary limited interaction. This approach has a lower risk-level compared to a high-interaction honeypot, while still being sophisticated enough to attract attacks targeting a specific vulnerability.

The classical example of such a honeypot is IoTPOT [27], which is a Telnet honeypot. This approach uses two components: a *Frontend Responder* which handles incoming connections, banner interactions, authentication and known command interactions. Unknown command interactions are forwarded to the IoTBOX backend that handles these requests and forwards them to the *Frontend Responder* that forwards the reply to the user.

High-interaction honeypots High-Interaction Honeypots provide an attacker full access to the system. This generates more information and allows for more elaborate analyses. The disadvantage of such an approach is that these systems are also easier to attack and abuse to launch DDoS attacks. To prevent this, a researcher deploying this should take additional security measures as discussed by Sokol and Andrejko [28].

An example of a high-interaction honeypot is SIPHON [12]. Their solution leverages 'wormholes' to connect physical IoT devices at one location to multiple endpoints around the world. Ceron et al. [13] proposed a high-interaction honeypot specifically for MikroTik RouterOS. Their solution showed that critical vulnerabilities in RouterOS were actively being abused, with an estimated two million devices still vulnerable to at least some of these vulnerabilities.

2.2.2 IoT firmware re-hosting

While there have been several attempts to use physical devices as high-interaction IoT honeypots, such as SIPHON [12], these solutions still require physical devices for each type of firmware. This quickly becomes unaffordable and difficult to scale in scenarios where many different types of firmware need to be analyzed.

One possible solution is *firmware re-hosting*. This is the process in which firmware is migrated from its original hardware environment into a virtualized one [29]. These

virtualized firmwares can then be run as a honeypot in full-system emulators, such as QEMU [30]. Firmware re-hosting also enables advanced dynamic and static analyses with solutions such as ANGR [31] and PANDA [32].

The advantage of using firmware images is that many manufacturers of IoT devices or routers publish firmware images on their website. Being able to re-host these images on regular hardware makes it trivially easy to deploy honeypots of all kinds of firmware and even allows for advanced vulnerability analysis. The firmware images can generally be divided in two groups: Linux-based firmware and monolithic firmware. Linux-based firmware is usually based on a conventional operating system in which code, data and libraries are separated. Monolithic firmware, on the other hand, intermixes all of these and usually does not depend on a conventional Operating System [33].

There are some different solutions that attempt to solve this problem of firmware re-hosting. We will discuss some of the state-of-the-art solutions to allow re-hosting of Linux-based and monolithic firmware in the next sections. This still remains an open problem with challenges that remain to be solved. A robot vacuum, for example, may require custom hardware necessary for running the motors. Re-hosting this firmware may fail, because this hardware is missing.

Linux-based firmware FIRMADYNE, the framework by Chen et al. [34] is one of the best-known frameworks that attempts to create a fully automated method of re-hosting Linux-based firmware images. Their approach is based on extracting the filesystem with BINWALK [35]. They discovered that the majority of the firmware images included accesses to Non-Volatile Memory (NVRAM). They included a way to emulate NVRAM by intercepting calls to `const char *nvram_get(const char *key)` and `const char *nvram_set(const char *key)` to redirect these to a key-value store. Furthermore, the kernel is replaced with a custom kernel. A script is used to infer the network connectivity. The main disadvantage is the low number of network-reachable images. Only 8.5% out of 23.055 firmware images from 42 vendors were reachable from the network.

Recently, Vetterl and Clayton [36] extended upon FIRMADYNE to make the extraction and emulation of Linux-based firmware images more accurate. A custom kernel has been used that intercepts signals that instructs the firmware to continue running if certain values in the NVRAM cannot be found. Other extensions are reading kernel logs, iteratively updating NVRAM values and rewriting the kernel's network configuration when the inferred configuration does not work. Their approach increased the number of network reachable images by 413% compared to FIRMADYNE. A disadvantage is that their approach is not published under an open-source license and is only usable via their online platform [37].

The latest addition, FIRMAE [8], made a significant increase in the number of successfully emulatable images by analyzing the firmware images that failed to execute in FIRMADYNE. The improvements consist of adding boot sequences, fixing missing filesystem structures and other fixes. The network connectivity is improved by retrieving IP addresses from external DHCP servers when necessary. Furthermore, improvements in

NVRAM emulation and kernel version allowed them to fully emulate 79.4% of network camera and router firmware images from eight different vendors.

Monolithic firmware The first attempts to analyze monolithic firmware require the use of physical devices during execution, such as PROSPECT [38], SURROGATES [39] and AVATAR [40, 41]. Other industrial attempts are successful in emulating entire architectures [42], though the hardware models still have to be modeled manually. Deployment of these aforementioned solutions are therefore limited in cost and scalability.

The state-of-the-art solutions try to mitigate these concerns by attempting to re-host firmware without requiring physical hardware during execution. The first attempt taking this approach is PRETENDER [29]. Their approach is based on AVATAR^2 [41] and tries to gather data from the real device to create models that are executable in an emulator. Still, this approach requires a physical device to record memory accesses and interrupts. Furthermore, the memory layout of the device needs to be known.

HALUCINATOR [43] attempts to remove the physical device by developing high-level replacements for Hardware Abstraction Layers (HALs) to allow re-hosting. This approach still requires a lot of manual configuration.

2.2.3 Fingerprinting honeypots

Another aspect to consider is the ability to fingerprint a honeypot. When an adversary can easily detect a honeypot, then they can simply ignore those devices. Vetterl and Clayton [44] have shown that many low-interaction honeypots can easily be detected, by observing slight differences in implementation compared to the original protocols.

Honeypots based on emulating firmware images can be detected in several ways. Brengel et al. [45] propose the use of timing attacks to detect hardware virtualized systems. Honware [36] used a similar approach to test the ability to fingerprint emulated devices compared to the real IoT and CPE devices.

There are some other approaches that can be used to detect virtualized systems, such as namespace detection and using user permissions [46]. Garfinkel et al. [47] mention discrepancies in CPU logic, I/O logic and the Translation Lookaside Buffer (TLB) as possible ways of detecting virtualization.

2.3 Automatic vulnerability detection

In the previous section, we discussed approaches on how honeypots can provide valuable insight on current attacks targeting IoT and CPE devices. High-interaction honeypots usually detect attacks using an Intrusion Detection System (IDS) [12, 13] and need signatures to detect intrusions. This section will discuss how Taint Analysis, Symbolic Execution and Control-Flow graphs can be used to automate the detection of vulnerabilities and zero-day attacks.

2.3.1 Taint analysis

Taint analysis can be used to follow suspicious data from taint *sources* to taint *sinks*. A *source* is the place where taint is introduced. Examples of these could include user input or network input. A *sink* is a location in the code where a user wants to perform some checks on the taints of certain variables [48]. Inputs at the source are marked as *tainted* and are followed during runtime until a sink is encountered.

Consider the following code example that allows an attacker to jump to an arbitrary line:

```
1 x := get_input()
2 y := 5
3 if x == 5 then goto 4 else goto 6
4     y := x+get_input()
5     goto y
6 print(y)
7 end
8 //unreachable code
```

We will use the *jump target taint policy* and operational semantics from SimpIL [49] to analyze this code example. There are different policies that can be used, each with their respective side effects. The jump target policy assumes that an input-derived value should never overwrite a control-flow value such as a return address or pointer. Other policies include the *tainted address policy* where a memory cell is tainted if either the memory cell or address is tainted.

One of the premises of the policy is to check whether or not a goto is tainted. The execution of the program then halts if one of the premises fails. Table 2.1 shows an example execution of the example code using taint analysis.

Line #	**Statement**	Δ	τ_Δ	**Rule**	**pc**
	start				1
1	$x :=$ *get_input*()	$\{x \to 5\}$	$\{x \to \mathbf{T}\}$	T-Assign	2
2	$y := 5$	$\{x \to 5\, y \to 5\}$	$\{x \to \mathbf{T}\}$	T-Assign	3
3	if $x == 5$	$\{x \to 5\, y \to 5\}$	$\{x \to \mathbf{T}\}$	T-Cond	4
4	$y := x +$ *get_input*()	$\{x \to 5, y \to 8\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-Assign	5
5	goto y	$\{x \to 5, y \to 8\}$	$\{x \to \mathbf{T}, y \to \mathbf{T}\}$	T-Goto	*error*

Table 2.1: Taint calculations for the example program using the SimpIL semantics

Taint analysis needs to carefully consider when to introduce taints and when to remove them. *Overtainting* or *undertainting* can easily occur when not all sources are considered carefully. An example of overtainting can occur in the following instruction in the *x86* architecture as shown by Schwartz et al. [49] and Portokalidis et al. [15]:

$$b = a \oplus a \tag{2.1}$$

An *xor* operation with the same register is used in abundance by *x86* to introduce zeroes as an alternative to the missing zero register. Keeping a taint on this instruction will

overtaint the data. Furthermore, it is cryptographically difficult to find an arbitrary value from a secure hash function. Hence, this output can be marked as untainted.

Another aspect to keep in mind is that this approach warns when tainted data is used in an unsafe way and not when the intrusion happens. An example of this is when an adversary leverages a buffer overflow to rewrite a return address. The alert only happens when the overwritten return address is called, which does not necessarily happen directly after the intrusion. This is called the *time of detection/time of attack gap* [49].

Dynamic and static approaches Taint analysis can be used in dynamic and static contexts. In a dynamic context, the program execution is followed and monitored. A disadvantage of taint analysis in a dynamic context is the added cost of execution. The PANDA framework reported a slowdown of 70x with taint analysis enabled in their original version [32], due to the need to analyze each instruction. Other frameworks that allow dynamic taint analysis are DECAF++ [50] and DROIDSCOPE [51] for taint analysis on Android-based systems.

A different method that can be taken is static taint analysis. Instead of following the program during execution, the code is analyzed and tainted statically. This approach requires less system resources and has been used by Cheng et al. [52] and Redini et al. [9] to find vulnerabilities in firmware images.

2.3.2 Symbolic execution

Instead of only following the execution during program analysis, symbolic execution allows for reasoning about the behaviour. This is done by building a logical formula that represents the program execution. Using this formula, different evaluation paths can be considered to discover which inputs trigger certain outputs [49]. The best known tools that use this approach are KLEE [53] and ANGR [31]. Their approaches define user input as a symbolic variable without any constraints. When a branch is encountered, a formula is constructed to constrain the possible values for the symbolic value. These constraints allow to reason what possible inputs can be used to trigger unvisited branches or possible exploits.

There are multiple search strategies that can be employed using Symbolic Execution. The most common are the traditional depth first strategies, heuristics and concolic testing. In concolic testing, the symbolic execution follows the same path as a trace of a program execution [49]. Symbolic execution is also used to generate Control Flow Graphs of the possible executions [31].

As an example, consider the following code:

```
1 x := get_input()
2 if x * 2 - 5 > 50:
3     x -= 10
4 else:
5     x := x * 2
6 if x == 25:
```

```
7     print("Success!")
8 end
```

To win the game above, the adversary needs to print the 'Success!' flag, which can only be achieved when x is chosen as 30.

Lets assume a depth-first approach is taken. At the first line, x will be set to a symbolic value s. The constructed formula for the first if-statement becomes: $[s * 2 - 5 > 50]$; the else statement requires the negated variant. Each newly discovered jump will append another constraint, hence the second if updates the constraint to: $[s * 2 - 5 > 50] \wedge [s - 10 == 25]$. This can be passed to a constraint solver to find the values that s should be 25 to win the game.

The multi-binary vulnerability discovery tool KARONTE [9] uses a similar symbolic path exploration technique in combination with static tainting in order to discover vulnerabilities in multi-binary firmware images. Their search heuristic is based on variables of interest. This means that paths that contain tainted variables are potentially more interesting and are prioritized.

2.3.3 Control-flow graphs

As briefly mentioned earlier, a good way to understand how certain vulnerabilities can be triggered is to map the different possible flows of the execution in a Control-Flow Graph (CFG). This is done by either using static analysis to generate a theoretical model of the indirect jumps, function data and other metadata. Another approach uses a dynamic model that resolves certain transactions that can only be resolved during program execution [31].

In such a graph, each of the basic blocks represents a set of instructions in which the execution begins from the first and terminates at the last. Each of the edges represents the different branches. There are also certain designated blocks: entry blocks allow control to enter the flow and exit blocks allow control to leave as discussed by Yousefi et al. [54].

Understanding the flow of a program can help in understanding in what ways a vulnerability can be triggered and can therefore be helpful to create signatures for the discovered vulnerabilities in combination with a statement solver.

Consider the following code:

```
1 x := get_input()
2 if x > 50:
3     print(f"The number {x} is larger than 50")
4 else:
5     print(f"The number {x} is smaller than or equal to 50")
6 end
```

This example has four basic blocks. First, the entry block contains line 1 and 2. The second block contains the True condition at line 3 and a block with the False condition at line 5. Finally the exit block contains the exit condition. The CFG then starts at block

1 with connections to the second and third block. Those blocks then connect to the final exit block.

2.3.4 Taint-style vulnerabilities

Another aspect to consider is the types of vulnerabilities that can be detected. The most important category of vulnerabilities to consider in high-interaction honeypots are the *taint-style* vulnerabilities. These are all types of vulnerabilities that are triggered when input data unsafely propagates through the system [52]. An example of such a vulnerability is Heartbleed, which is a vulnerability in the OpenSSL extension Heartbeat. The extension did not properly check if the provided payload length is the same length as the **HeartbeatRequest**. By specifying an incorrect payload length an adversary could read up to 2^{16} bytes of private memory, potentially exposing cryptographic secrets [55].

Unsafe methods in programming languages, such as C, are another potential source of taint-style vulnerabilities. Take the *strcpy()* method, which copies a string into a buffer. The method does not check if the string fits inside the destination buffer and could therefore cause a buffer-overflow error. The replacement method *strlcpy()* is a safer replacement that only copies a maximum number of bytes and null-terminates the copied string [56]. Newer low-level programming languages employ different strategies to decrease the number of such vulnerabilities. The Rust language uses a borrow-checker to guarantee memory and thread-safety. This language has since been used to replace critical components in the Firefox browser engine [57]. These programming languages can help provide a higher level of security and reliability to IoT devices as shown by Dwelo, a company that has rewritten to code of their IoT gateways in Rust [58].

2.3.5 Signature generation

As discussed earlier, IoT devices are usually only supported for a short time. When a vulnerability has been found, many devices will remain unpatched and vulnerable. A signature that characterizes a vulnerability can help to identify malicious traffic and protect end-users by blocking this traffic. Many solutions proposed by researchers involve blocking malicious traffic using vulnerability signatures, such as DeadBolt [14].

A vulnerability can be triggered in multiple ways. A signature should therefore be based on the properties of the vulnerability and not on a single exploit, as discussed by Chen et al. [34]. They proposed a formal definition of generating signatures based on the data-flows in the program that takes the vulnerable inputs and execution paths that exercise a vulnerability into account.

Furthermore, the *completeness* and *soundness* of a signature are other aspects to consider. An incomplete, but sound signature will produce false negatives. A complete, yet unsound signature produces false positives and may block regular traffic as well. The approach of Chen et al. prioritizes soundness, but they provide a method to reformulate their algorithm to prioritize complete signatures.

Chapter 3

Approach

The goal of this research is to evaluate the use of static analysis techniques to model IoT device firmware and to drive vulnerability detection in a high-interaction honeypot running this firmware. The literature research in Section 2 identified several frameworks for static analysis and several frameworks that can be used to re-host device firmware in honeypots. In order to reach this goal and to answer the research questions as defined in Section 1.1, we defined the following requirements for a prototype system:

Requirement 1 *The static analysis must be able to detect sources of taint in a firmware image.* This solution targets taint-based vulnerabilities, which means that the vulnerability must depend on user input. The system should therefore be able to distinguish user-based inputs from other inputs.

Requirement 2 *The static analysis must use symbolic execution to track constraints on tainted variables and check if user input can trigger a vulnerability.* With symbolic execution, we can construct a formula that can be used to verify if the vulnerable method can be reached with the collected user input.

Requirement 3 *The honeypot must allow for debugging on the firmware binaries at the detected input locations to collect user inputs.* The system will be deployed as a honeypot, which means that it should be able to collect the user inputs from the vulnerable input sources from **Requirement 1**.

Requirement 4 *To verify if an input can reach and trigger a vulnerability, the system must be able to reconstruct a symbolic execution state.* To verify if a user input can reach and trigger a vulnerability, we need to reconstruct the symbolic formula from the static analysis. The solution needs to store all relevant information from the vulnerability and user inputs to make it possible to reconstruct the formula and verify satisfiability.

Requirement 5 *The system should at least work for Linux-based firmware images on the ARM32 architecture.* Each architecture works slightly different, which means that the

system needs to be adapted for each additional architecture that is supported. For this reason, we at least require support for the commonly used ARM32 architecture.

Three components are used to meet these requirements. The components are depicted in Figure 3.1. The first component is the Firmware Analyzer which identifies vulnerable input sources of taint. Secondly, the honeypot collects user inputs at the identified input sources. Finally, the Input Analyzer reconstructs a symbolic execution state to verify if the input can trigger a vulnerability.

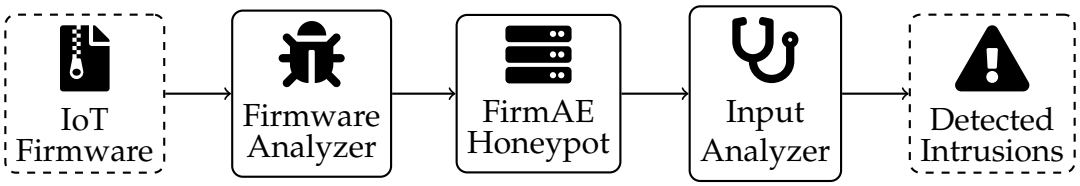

Figure 3.1: Schematic overview of the system components.

The Firmware Analyzer is based on Karonte [9], since this framework provides a combination of static taint analysis and symbolic execution to detect multi-binary taint-style vulnerabilities. Karonte is based on Angr [31] and supports the detection of three vulnerability types: buffer overflows, loops guarded by tainted data and tainted call addresses. To answer Research Question 1, we determine how the input sources in the honeypot are chosen.

The high-interaction honeypot is based on FirmAE [8], since their solution allows for easy re-hosting of Linux-based firmware. The honeypot is adapted to collect the inputs for each of the user interactions. The collected inputs are used to reconstruct the execution state in Karonte to identify if the inputs can trigger certain vulnerabilities. FirmAE is based on QEMU, which allows for fine-grained monitoring of system activity and also allows for external debugging with GDB. To answer Research Question 2, we discover how we can reconstruct the execution state in order to determine if the collected inputs from the honeypot can reach and trigger a vulnerability.

The main architecture that our prototype supports is ARM32. Karonte and FirmAE both have support for ARM32- and MIPS-based firmware images, although support for MIPS is limited in Karonte [9]. Similarly, FirmAE and Karonte only support Linux-based firmware. Other solutions that also target monolithic firmware require much more manual configuration and are currently not suitable for use in this research. For this reason, this research only considers Linux-based firmware images. FirmAE is verified against firmware images for routers and smart cameras. Extending FirmAE to support other device types is not a goal of this research and for this reason, the prototype only supports routers and smart cameras.

In order to answer Research Question 3, we deployed honeypots based on the prototype. The alerts collected by the honeypots are analyzed to answer RQ3. When no attacks are detected in the honeypots, then the analysis are extended to the vulnerabilities and exploits from the original Karonte dataset. The performance of the alerts are analyzed using performance metrics. Each honeypot contains an IDS module to analyze the network traffic for attacks. To answer the final research question, we

analyze if the static analysis component provides benefit over a traditional IDS-based approach of detecting attacks by evaluating the performance of the IDS compared to the performance of the static analysis.

The described system has been implemented as a prototype. The activities for each component and the details of the implementation are described in more detail in the next chapter.

Chapter 4

Prototype

In order to test the the main question of this research, we created a prototype based on KARONTE [9] and FIRMAE [8]. This prototype consists of three major components: A firmware analyzer, a honeypot based on FIRMAE and an input analyzer. This section describes the design of each of these components.

4.1 Firmware Analyzer

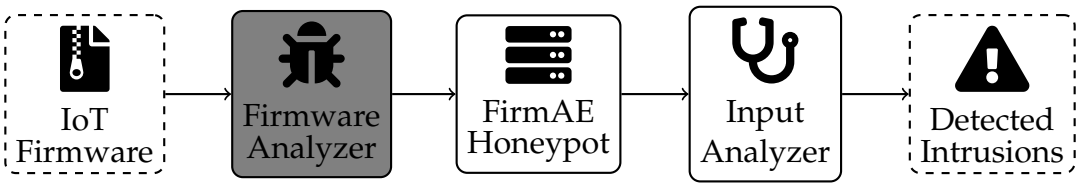

Figure 4.1: Schematic overview of the system components.

First, we explain the working of the Firmware Analyzer. The firmware analyzer takes a firmware image as input, analyzes the image and returns two outputs: A file containing alerts, all the necessary information related to the alerts and a file containing input sources related to the alerts that can be parsed by the honeypot.

As mentioned earlier, the Firmware Analyzer is based on the KARONTE framework. This framework works with three analyses: The Border Binaries Finder, Binary Dependency Graph and the Bug Finder. Figure 4.2 shows an overview of all the major components of the Firmware Analyzer. For this research, some major changes have been made to the KARONTE Framework. In this section, we discuss the working of each individual component of the Firmware Analyzer and all modifications that have been made.

4.1.1 Extractor

The first step in the Firmware Analyzer is to extract the firmware image. This step uses BINWALK [35] to recursively extract the first compressed file system from a Linux-based firmware image.

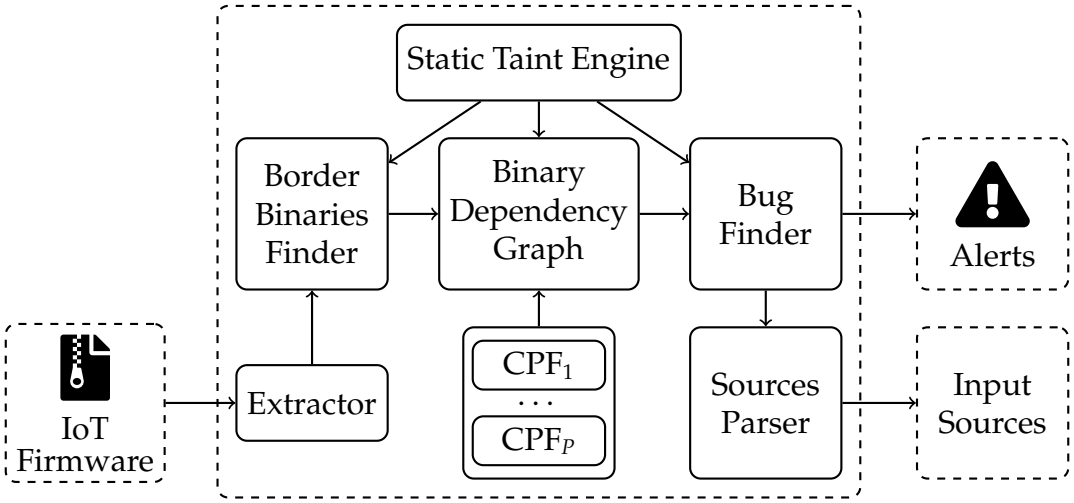

Figure 4.2: Schematic overview of the Firmware Analyzer

The original version of Karonte used a shell script to extract all available file systems and files up to a certain depth. To provide consistent results with the extractor in the honeypot and the Firmware Analyzer, we replaced this module with the extractor module from FirmAE, which utilizes the Binwalk Python API [35]. This module has been modified to remove the extraction of the kernel and the database connectivity.

This change has three advantages: First, we can guarantee that the analyzed file system by the Firmware Analyzer is exactly the same as in the honeypot. Secondly, this module has better error detection for firmware images and can skip blacklisted files. Finally, only the file system is extracted, instead of all folders inside the archive. This ensures that the folder it extracts to only contains the file system and not other files, such as documentation and manuals, with the file system extracted in a subdirectory.

4.1.2 Border Binaries Finder

When the file system is extracted, we can start analyzing the firmware. The goal is to detect taint-style vulnerabilities that are exploitable over the network. To do so, the Firmware Analyzer needs to discover which binaries in a firmware image have network-services. These binaries are the network-facing binaries, which represent the entry-point of the vulnerability analysis. The Border Binaries Finder module attempts to discover these binaries by identifying functions in binaries that *parse* and *receive* user data.

The Border Binaries Finder iterates over each function in a binary and scores each function using five features that are used to calculate the parsing score of a function [9]:

1. The number of basic blocks (#*bb*)
2. The number of branches (#*br*)
3. The number of memory comparisons (#*cmp*)

4. The *network mark* (#*net*)
5. The *connection mark* (#*conn*)

The network mark is a multiplier that is used to identify the number of memory comparisons that handle network strings. This is done by comparing the memory location of the referenced strings with a list of network strings. The *network mark* is incremented for each network string used in a memory comparison.

The connection mark is a second multiplier that is added when a data-flow from a socket (*read* or *recv* methods) can reach into a memory comparison. This is checked by leveraging a forward taint analysis from the socket or a backward taint-analysis from the memory comparison.

The parsing score of (ps_b) of a binary b is then calculated using the following formula by combining the five features, as shown in Equations 4.1 and 4.2:

$$ps_b = max(\{ps_j | \forall_j \in get\_functions(b)\}) \tag{4.1}$$

$$ps_j = (\sum_{i \in bb,br,cmp} k_i \cdot \#i_j) \cdot (1 + k_n \cdot \#net_j) \cdot (1 + k_c \cdot \#conn_j) \tag{4.2}$$

The parameters for k_{bb}, k_{br}, k_{cmp}, k_n and k_c are defined in the KARONTE paper [9]. This formula generates a score for each binary. Since most binaries have at least some score, we need to distinguish the most significant binaries. KARONTE uses DBSCAN for this, which groups closely scoring binaries together. The binaries in the highest scoring cluster are considered the Border Binaries. The list of the border binaries, all binaries with a parsing score and the information regarding the five different features are written to disk.

Modifications We only made several small changes to the Border Binaries Finder. The changes that we made to the Border Binaries Finder were almost all performance improvements to increase performance of this module.

The first upgrade is to use multi-processing. All the analysis done by this module are single-threaded. By running the calculations for each binary in a different process, we can significantly speed up the calculations for this module.

Another upgrade is to check the necessity to do the full analysis. The parsing score is always zero when there are no memory comparisons inside a binary. The upgraded version checks if there are memory comparisons inside a binary and only calculate a control-flow-graph to calculate the parsing score when there is at least one memory comparison. Calculating a control-flow-graph is a memory intensive task and should be avoided if possible.

Similarly, the *connection mark* is now only calculated when there are sockets and memory comparisons available. Previously, the taint analysis was also run when there was no socket to taint traffic from. In such a case, it is impossible to have taint travel from a socket to a memory comparison. Therefore it is unnecessary to run this computationally expensive step of the analysis.

4.1.3 Binary Dependency Graph

The border binaries can now be used to discover the communication between binaries, which is later used to discover multi-binary vulnerabilities in the Bug Finder. The KARONTE framework can detect either single- and multi-binary vulnerabilities. Multi binary vulnerabilities are using Communication Paradigms to communicate information between binaries. An example of this is one binary setting an environment variable for the key: *"QUERY_STRING"* and a second binary that retrieves the environment variable for this same string. The goal of the Binary Dependency Graph (BDG) module is to detect these communication patterns and build a graph that connects all the binaries.

The binary dependency graph module works by first identifying all possible basic blocks leading into function calls referencing network keywords in argument registers. These network keys are called *Data Keys*. A taint analysis is then started on the function in which the data key is referenced and taint is applied to the network keyword. The taint analysis module uses symbolic execution to explore the individual in the graph for a set period of time. The Communication Paradigm Finder (CPF) module is run on each basic block it analyzes.

Communication Paradigm Finder The CPF modules are used to detect if the examined code path has the required abilities to share data through the functionality that the CPF describes. There are five CPFs currently available in KARONTE:

1. Environment: *setenv* and *getenv* functions
2. File: communication via files
3. Socket: communication over network sockets
4. Setter_Getter: Function calls to functions containing either *set* or *get* in the function name
5. Semantic: Detects tainted data used as index, indirect data accesses or tainted data copied to global structures

The CPF is run on each of the basic blocks that is analyzed and checks if the basic block corresponds to the CPF method. For example, in the Environment CPF, we check if the block contains a call into either *getenv* or *setenv* and if one of the argument registers contains a reference to a data key.

Besides detecting communication patterns, the CPF module detects the *Role* of a binary for this data key. This can be either a *setter* or a *getter*. Setter binaries are used to propagate data to other binaries and getter binaries are used to retrieve and process data. The role information is stored in combination with extra information regarding the function, basic block, role function and the register that contained the data key. This information is later used to construct the dependency graph. This information is called the *role information* and is used later to initialize the vulnerability analysis. Secondly, the CPF detects other data keys that are depending on the same CPF by analyzing all the basic blocks containing a call into a function belonging to the CPF. Finally each

CPF contains functionality to detect new binaries that refer to the same data key. These binaries are then scheduled for further analysis.

After analyzing all binaries for possible roles, the BDG module reconstructs the graph representing all the communication between binaries. More on this reconstruction algorithm can be found in the KARONTE paper [9]. The BDG module returns a graph where each binary represents a node and it returns for each binary a dictionary with the discovered role and relevant information related to the role.

Modifications Some modifications were made to the BDG and CPF modules. First, there are two ways that strings can be referenced: direct string references, where the location of the string is directly referenced in an instruction (in the VEX intermediate representation) as shown in Equation 4.3:

$$\text{PUT(offset = 8) = 0x00000008} \tag{4.3}$$

and indirect string references, where the location of the string is referenced indirectly, for example via a pointer located on the stack (Equation 4.4: also in the VEX intermediate representation):

$$\begin{aligned} t4 &= \text{LDle:I32(0x0000d52c)} \\ \text{PUT(offset = 12)} &= \text{t4} \end{aligned} \tag{4.4}$$

The algorithm to detect indirect string references was deprecated due to the use of deprecated functionality in ANGR. We have rewritten this algorithm to detect these accesses using higher-level abstractions provided by ANGR. ANGR has an XREFmanager that can detect indirect string cross-references. This is done by iterating over all the functions in a binary. For each basic block in a function, it verifies if the block ends in a *call* instruction. From these basic blocks, the algorithm extracts all instructions that set a value to an *argument register*. Each instruction is then checked for indirect string cross-references using the XREFmanager and the constant field of the instruction is used to discover direct string references.

Next, in the original version the BDG started the taint analysis at the block where the data key was discovered, instead of at the start of the function. This has the advantage that taint analysis is faster, since all preceding blocks do not have to be analyzed. A problem with this approach is that the stack is not initialized properly, because the other blocks are not executed. An indirect string reference using a pointer on the stack does not work as expected under this approach, since this pointer on this location in the stack would not be initialized and the CPF module would then not be able to detect a data flow using this data key. To solve this, the taint analysis is now started at the start of the function to ensure that all variables during the execution of a function are initialized properly.

Another change is made to the discovery of new binaries to be analyzed. There were inefficiencies slowing down the analysis and several modifications have been made to speed this up: First, when a binary is already discovered and added to the list to analyze, it is not be considered again at a later stage. Some CPFs also have additional

checks to ensure that an added binary is actually a getter of that data key. This is done by generating a CFG of that binary and checking if the data key is used in an argument register in a call to a getter function. Instead, we first check if an entry exists in the Procedure Linkage Table (PLT) for a possible getter function and only then consider this binary as a potential candidate and generate a CFG.

4.1.4 Bug Finder

With the Binary Dependency Graph ready, the bug finder analysis can be used to detect vulnerabilities in the binary. The Bug Finder (BF) module can perform two types of analyses: A multi-binary vulnerability analysis and a single binary vulnerability analysis. The multi binary vulnerability analysis is the default and when no vulnerabilities are discovered, the single binary analysis is used.

The multi-binary vulnerability analysis starts at *parent* binaries. These are the binaries in the top of the Binary dependency graph that communicate to *child* binaries. The module uses the *role information* that was discovered by the Binary Dependency Graph module and starts a taint analysis at the start of the function in which the role was detected. Then a taint analysis and symbolic execution is started at the start of the function, which analyzes each execution path. For each basic block in the execution path, we first check if the block is the block in which the role was detected. If this is the case, taint is applied to all function registers in this basic block. Secondly for each basic block, we do the vulnerability analysis. This is explained later in this section. The vulnerability analysis is performed from the parent binaries to record the propagated parameters using Summary Functions that are 'wrapped' around library functions (more on this in Section 4.1.5). After the parent binaries are executed, the child binaries are analyzed. Finally the *orphan* binaries are analyzed. These are the binaries in which communication patterns were detected, but no binary was detected to which it communicates.

The single binary vulnerability analysis is called after the multi-binary analysis and is only executed when nothing is found during the multi binary analysis. This analysis discards the entire BDG and all the roles. It uses a similar method as in the BDG class to detect network keywords in function calls. Instead of using CPF modules it verifies if network keywords are used in *memcmp*-like functions (this includes strcmp, *strcasecmp*, *strstr*, etc.). This also defines new *role information* with the role *UNKNOWN*. A similar vulnerability analysis is then started at the start of the function in which the *memcmp*-like function was detected.

Vulnerability Analysis The current version of the Bug Finder module can detect three types of vulnerabilities: buffer overflows, loops depending on tainted variables and tainted function calls. Each of these types of vulnerabilities have different checks to verify if a vulnerability can be triggered.

First, we have the buffer overflow vulnerabilities. All buffer overflows that we check are sink-based vulnerabilities. This means that we verify if the block leads into a *sink* function. This is a potentially vulnerable function. When a sink is encountered, we

verify if there is taint in the registers. This depends on the function that is analyzed. For *strcpy*-like methods, we only need to verify if there is taint in the source register. For *memcpy*-like methods we need to verify if there is taint in the register that defines the size to copy.

When the existence of taint in the required register is verified, we need to verify if a vulnerability can be triggered. In the example of a *strcpy*-like sink, we need to verify that the size of the source buffer is larger than the destination buffer. The rules to check for a vulnerability have been updated and improved. For example, in the original version, there was no verification for the size of variables for *sprintf*-like methods. This functionality has been added to prevent these functions to be incorrectly identified as vulnerable.

Storing vulnerability information Information regarding vulnerabilities is stored in two places. First, an entry is written to the log file, which provides some basic information on the vulnerability. An example of such an entry can be seen in Figure 4.3.

To be able detect inputs in the honeypot, if an input can reach a vulnerability and if the inputs can actually trigger a vulnerability, we also need to store additional information. As mentioned earlier, the vulnerability detection is based on symbolic execution. ANGR constructs a symbolic formula during the execution that can be used to determine satisfiability of the execution path. We also need to store the names of the registers to which we applied taint including the symbolic variables stored in these registers.

Furthermore, information on the sink is stored, including the contents of the argument registers. In case of a tainted loop or tainted function calls, this information is replaced with the information necessary to verify if this vulnerability can be triggered. Finally, we store the *role information*, the type of CPF that has been used to detect the vulnerability and the name of the binary.

The information of the role is used later to determine the locations the honeypot uses to collect inputs (more on this in Section 4.1.6). The stored information is also used in the Alert Analyzer to check if an input can reach and trigger a vulnerability. This is explained later in Section 4.3.

4.1.5 Taint Analysis

The previous three modules all depend on taint analysis. In this section, we explain some of the details behind the taint analysis module. The taint analysis uses a state-based exploration technique. The code is converted into a Control Flow Graph consisting of basic blocks. The symbolic execution explores each path on a state-by-state basis. During this a symbolic formula is constructed, as explained in Section 2.3.2.

The taint analysis module can be provided with a custom method that is called every time a basic block is visited. This can be used to define when taint needs to be applied and to analyze the current state and instructions in the basic block, as is done in the Bug Finder module.

```
====================== Start Info path ======================
Binary: httpd:
Plugin responsible to propagate the data: UNKNOWN
Key: http_rpm_update, Sink address: 0xd4c0, time: 2.33701229095459 sec

                              Path
                              ----
0xd374 -> 0xd3b8 -> 0x9630 -> 0xd3c0 -> 0xd3dc -> 0xd3e8 -> 0x9864
-> 0xd3ec -> 0xd404 -> 0x9630 -> 0xd424 -> 0xd450 -> 0xd47c ->
0xd490 -> 0x98d0 -> 0xd4a4 -> 0xd4ac -> 0x957c

                        Fully tainted conditions
                        ------------------------
====================== End Info path ======================
```

Figure 4.3: Example log entry of a buffer overflow in the TP-Link Archer C3200

Summary functions Some library functions are wrapped by the Taint Analysis module and are executed before the block is analyzed. Most library functions are replaced by Simulated Procedures, which are high-level replacements of library functions. These allow for better tracking and propagation of constraints. The Summary Functions are used to extend this functionality with the following capabilities:

- *Untaint variables when variables are constrained* The option to untaint variables can be used when variables become constrained. An example of this is the *memcpy* method. When the *n* argument is tainted, there is a possibility of having a buffer overflow. When this is not the case, we untaint the buffer that the variable is copied into.
- *Propagate taint* This is discussed in more detail in the next paragraph.
- *Fix small mistakes in the Simulated Procedures* the Simulated Procedures have some assumptions to speed up the analysis. A call to *malloc* assumes the maximum available size when the parameter is symbolic. This works well when you do not have to reason about size, but becomes problematic when this is a requirement. The Summary Functions fix this by replacing the Simulated Procedure with a custom replacement of this method. Another small update is that the symbolic variable and the name of the return register are stored and returned to the Bug Finder module for functions where the return value is relevant to the symbolic formula. This is done to ensure that the Honeypot and the Alert Analyzer modules also consider the return value when determining if inputs can trigger a vulnerability.

Taint and Constraint Propagation There have been two significant changes in the Taint Analysis module. First, the way of propagating taint and constraints has been changed. Previously, the taint was propagated using the Summary Functions module.

Each summary function had two options during execution. Either the Simulated Procedure was executed or a 'FakeReturn' was inserted in the block jumping into the library function to skip the function call. Then the output or destination buffer of the function are replaced with a tainted bit vector. The second option was mostly used when one of the argument registers contained taint.

This approach replaces the output with a tainted bit vector that is unconstrained. One of the requirements for this project is that we should be able to accurately detect if a path is reachable. When the execution path depends on this variable, as shown in the code example below, then there is no way of determining anymore which of the two execution paths are reachable:

```
1 int network_copy(char network_input[]) {
2     dst_buffer = (char *) malloc(1000);
3     // test if the network_input starts with a data key
4     if memcmp(network_input, "http_referer", 12) {
5         strcpy(dst_buffer, network_input)
6     }
7     return dst_buffer
8 }
```

In this case, the vulnerability can only be triggered when the network input starts with 'http_referer'. Not copying constraints when the network input is tainted would cause the output of the memcmp to be unconstrained. Verifying a string for this vulnerability path would mean that any input would be able to reach the vulnerable strcpy, instead of only input starting with the data key.

To solve this, we have rewritten the summary functions to always execute the Simulated Procedures. The bit vector in the output is afterwards replaced with a tainted bit vector and stored into memory. All the constraints from the Simulated Procedure are copied over to the new bit vector. Using this approach, we can assure that the taint and the constraints on variables are copied over correctly.

Buffer Size Analysis Another significant change in the Taint Analysis is the way to determine the allocated size of buffers. Previously, the analysis always applied 4096 bytes of taint at the location of the tainted input, regardless of the allocated size. During the symbolic exploration, when a sink is encountered, it used the *Size Analysis* module which started a taint analysis in order to discover an allocation instruction and discover the allocated size of this variable. Because of this, only sizes of heap variables could be discovered. Furthermore, it required additional taint analysis which slows down execution time. Another problem with this approach is that always tainting 4096 bytes has the potential to cause overtaint, where unrelated neighbouring variables become tainted, which has the potential to incorrectly identify these input sources as vulnerable.

This Size Analysis module has been replaced with new functionality to discover allocated size based on abstractions in Angr, the underlying framework of Karonte. This approach can detect the size of stack and heap variables and does not require additional

taint analysis. The memory layout in ANGR uses Bit Vectors. There are two types of bit vectors: Symbolic Bit Vectors (BVS) and concrete bit vectors (BVV). A BVS can still assume any value and can be constrained, while a BVV contains a concrete value. The stack and heap start empty and are only initialized when necessary to keep memory usage low as shown in Figure 4.4:

Figure 4.4: Symbolic Stack space at the start of the function

Stack and heap variables are initialized by placing empty symbolic bit vectors in the memory. An instruction allocating a string array of 100 characters would initialize an empty symbolic bit vector and store this in the memory used for symbolic execution as shown in Figure 4.5.

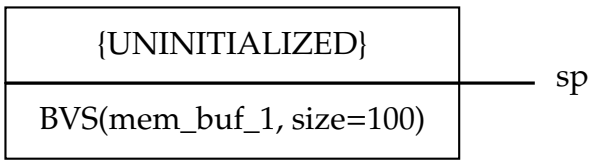

Figure 4.5: Symbolic Stack space after initializing one stack variable

Retrieving memory using these abstractions returns a bit vector consisting of multiple smaller bit vectors. The allocated size is retrieved by loading 4096 bytes into memory. From this, we extract the size of the first Bit Vector. The size of 4096 is chosen to keep the memory usage to an acceptable level, since multiple bit vectors need to be loaded and stored in execution states during the symbolic execution.

Using this approach, we can directly apply taint to only the allocated memory on the stack or heap and compare both allocated sizes when verifying if a sink can trigger a vulnerability. Similarly, when propagating taint, this module is used to estimate the size of the buffer which needs to be tainted.

4.1.6 Sources Parser

With vulnerable execution paths discovered, there is only one remaining step. The alerts need to parsed in such a way that these can be used inside the honeypot. Furthermore, we need to discover the location we use for colleting inputs, in order to collect the inputs in the honeypot and to answer **Research Question 1**. We have created the *Sources Parser* module, which takes the alerts from the Bug Finder and transforms them into input sources for the honeypot. Furthermore, it adds extra information necessary for the honeypot to run.

Determining Input Sources There are two different scenarios for input sources: a scenario for single-binary vulnerabilities and a scenario for multi-binary vulnerabilities.

Both of these base the input sources on the *roles* that are determined by the BDG module. In the single binary case the roles are determined from *memcmp*-like functions using network keywords. The path to reconstruct these needs to replace the tainted variables with the collected inputs.

1. *Single-binary vulnerabilities* To do so, the basic block containing a call into the role function is analyzed to determine the input sources for single-binary vulnerabilities. This is done by determining which argument registers are used in the function call to the role function.

   Then it returns a list of register names that the honeypot should collect inputs from, excluding the register containing the data key. This is returned in combination with the address of the basic block, the address of the call instruction and the address of the role function that is called. With this information, the honeypot can retrieve all the necessary inputs for this vulnerability.

2. *Multi-binary vulnerabilities* The determination of input sources for multi-binary vulnerabilities work mostly the same. The input sources are determined for the role in which the vulnerability was detected. These input sources are determined in the same way as for the single binary vulnerabilities. The reason that the full path through each binary is not taken into account is that this passes only the network input. To be able to discover if an input can reach and trigger a vulnerability, we need to know all parameters passed in the vulnerable function. For example, if the vulnerable method is detected using the *Semantic* plugin, with the *memcmp*-function, then the passed parameter also need to know the *n* parameter passed to this function. When collecting the network input at a higher level in the dependency graph, then we would not be able to discover which parameters would be passed to this function making it more difficult to accurately detect if a vulnerability can be triggered and reached. For this reason, the input sources are chosen in the binary in which the vulnerability is discovered.

   Finally, for multi-binary vulnerabilities we need to consider the return value for certain role function. In a call to *getenv*, for example, the information from the return register is required to reconstruct the execution path in the Alert Analyzer. Therefore, the Summary Functions module has been changed to also store the return registers and symbolic variables to which taint has been applied. These registers are then also parsed using this module and passed to the honeypot.

Symbolic Links Another aspect to consider when parsing the vulnerabilities for the honeypot are symbolic links. The file system of a firmware consists of multiple binaries and symbolic links. There may be multiple symbolic links linking to a vulnerable binary, while these may show up in the process list running under the name of the symbolic link instead. An example of this is the */usr/sbin/udhcpc* binary in the D-Link DIR-868L router. This binary links to */usr/sbin/udhcpd* and is one of the binaries instantiated at system boot and also appears with the symbolic link in the processes list.

To ensure that we can capture inputs from all binaries in the honeypot, this analysis prepares a map of all symbolic links to the destination binary and passes this to the

honeypot as well.

Position Independent Executables One thing to consider for the honeypot are Position Independent Executables (PIE). These binaries are built with the hardened security features of GCC. Instead of loading the binary and its dependencies in a static location, these binaries are instead loaded into random memory locations each time the binary is executed to provide a higher level of security [59].

Angr applies a default offset to Position Independent Executables and all memory addresses used in the analysis have this offset applied. In the honeypot we need to account for this offset and the random offset of a running binary. In order to do so, we provide the address of the *_start* function of each binary (*angr_start*). The honeypot can then use GDB to discover the start address of the running binary (*gdb_start*). The address of a vulnerable function in a running binary on the honeypot ($func\_addr$) can then be calculated using both start addresses and the address of the function discovered by Angr ($angr\_func\_addr$) as shown by Formula 4.5:

$$func\_addr = angr\_func\_addr - angr\_start + gdb\_start \tag{4.5}$$

4.1.7 Improvements in Karonte

There are some other general improvements that have been made to Karonte. In this subsection, we want to discuss these improvements.

Documentation of dependencies The Karonte framework was only runnable using an outdated Docker image and did not have any documentation on the dependencies used and the required versions. The problem is that breaking changes have been made to the underlying frameworks. To make the framework easily runnable without Docker, the dependencies have been documented with the required versions in a Python Virtual Environment. Setting up the codebase is now as easy as installing Binwalk and running the command to install the Python dependencies from the *requirements.txt* file. The prototype is currently based on Python 3.8 and Angr 9.0.5739.

Upgrade to Python 3 As mentioned in the previous paragraph, most of Karonte is running on outdated dependencies. Furthermore, it is also running on Python 2. This version has received the final update in January 2020 and is considered end-of-life.

In order to get more familiar with the framework, we spent time upgrading the Karonte framework to the latest version of Python and Angr. Time has also been spent in the upgrade to make better use of the available abstractions provided by Angr. An example of this are the Link Registers. Not all architectures have a dedicated link register and instead use the Stack to store the return address of a function. The upgrade abstracted away the use of link registers in most places to make Karonte easier to upgrade to more architectures in the future. The Summary Functions, for example, used link registers and 'Fake Returns' to skip over library calls to provide a custom replacement of the function and to propagate the taint. This has been replaced with the use of Simulated Procedures.

Furthermore, the code base contained multiple duplicate or unused methods. To increase readability and maintainability, these have been merged together or removed. Similar stylistic changes were made to increase readability as well. The other upgrades are smaller and are mostly done to increase performance and memory usage of the framework.

Finally, all changes to Karonte were contributed back into Karonte using a pull request[1].

4.1.8 General overview of modifications

Component	Modification
Extractor	Replaced extractor with FirmAE Extractor
Border Binaries Finder	Multi-processing
	Only calculate CFGs if necessary
	Only perform Taint Analysis if necessary
Binary Dependency Graph	New algorithm for indirect string references
	Taint analysis at function start
	Speed optimizations in CPFs
Bug Finder	Storing vulnerability information
	Improved sink rules
Taint Analysis	Buffer size analysis
	Taint and constraint propagation
Sources Parser	Everything (new)
Other	Upgrade to Python 3
	Documentation of dependencies
	Code cleanup and speed improvements
	Better use of Angr abstractions
	Contributed changes back into Karonte

Table 4.1: List of all modifications to Karonte

4.2 Honeypot / Input Collector

The vulnerabilities discovered by the Firmware Analyzer can be used to detect attacks targeting these vulnerabilities in this firmware. This is done by re-hosting the firmware using FirmAE and attaching debuggers to the emulated firmware. Section 2.2.2 of the related work discusses the process and recent developments in the re-hosting of firmware images. In this section, we discuss how we used FirmAE and the output of the Firmware Analyzer to create a honeypot that can collect inputs based on the input sources from the Firmware Analyzer.

[1] `https://github.com/ucsb-seclab/karonte/pull/13`

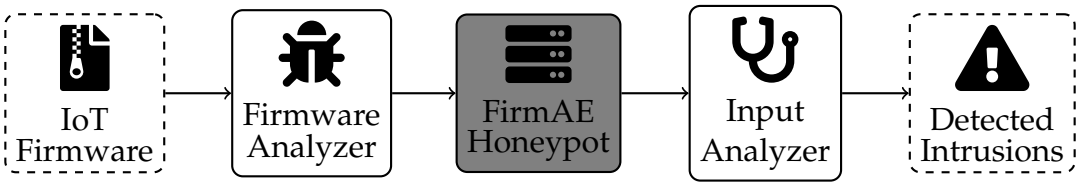

Figure 4.6: Schematic overview of the system components.

4.2.1 System components

The honeypot consists of three distinct components. Figure 4.7 shows a schematic overview of the different components inside the honeypot. To understand how the inputs are collected on the honeypot, we discuss each of these components in detail. The host system can be any Linux-based distribution and can be easily deployed in a cloud environment as we shown later in Section 5.

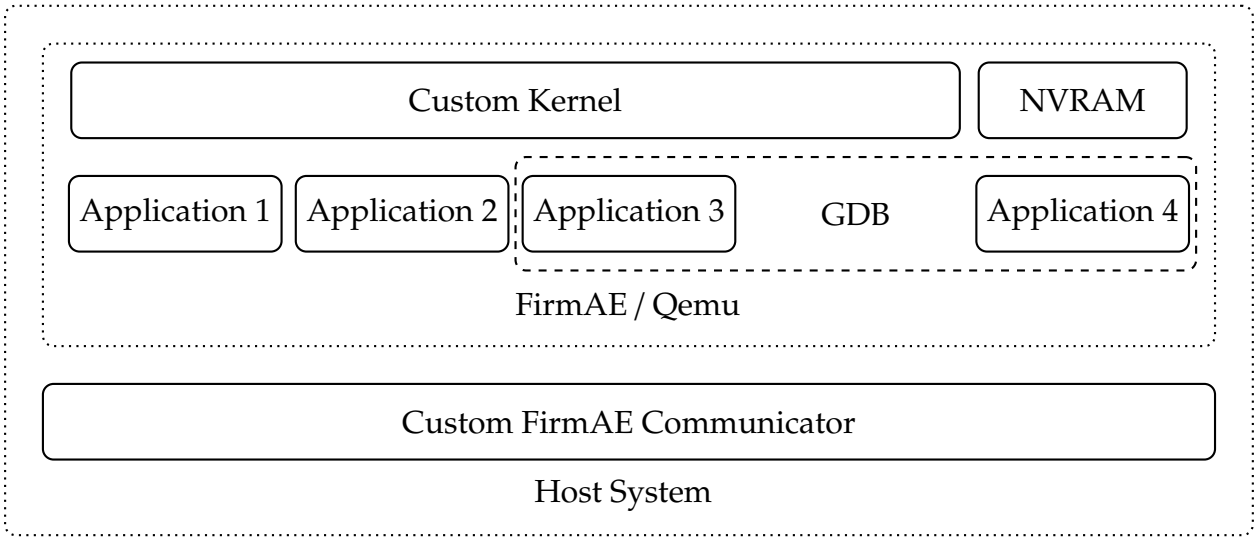

Figure 4.7: Schematic overview of the honeypot components

FirmAE/Qemu As mentioned earlier, the main component of the honeypot is FirmAE which emulates the firmware image. This is done by extracting the firmware image and replacing the kernel with a generic modified kernel that is modified to keep the system running outside the original hardware. It also adds emulated NVRAM, which is a necessary component that many firmware images rely on.

The inputs in the honeypot are collected using GDB. For each of the running applications that were identified by the Firmware Analyzer a GDB Server is started, which is later used to set breakpoints on the input sources to collect the possible vulnerable user inputs.

The honeypot is being run using the *debug* mode of FirmAE. This mode modifies the *preinit* of the binary to start a *netcat* session to the host system. The start script of FirmAE is modified such that we can pass a custom communication script which connects to this netcat port and can communicate with honeypot to control the GDB Servers on the

honeypot to collect the inputs. All inputs are collected directly on the host system to ensure that no input gets lost after a sudden reboot (which resets the entire honeypot to the original state)

Custom FirmAE Communicator The custom FirmAE communicator is responsible for managing the communications with the honeypot. The honeypot starts a *netcat* server. This communicator is started and attempts to connect to the netcat server on the honeypot. This script is responsible for determining which processes need debugging. This could either be the process itself or a process started via a symbolic link. For each of the identified processes the communicator sends a signal to open a GDB Server on the server and allow communication to the host on a custom port.

With GDB servers started, the communicator is also responsible to start GDB on the host system and to connect to the honeypot for each of the started GDB Servers. This process gets a debugging script that controls how the inputs should be collected for that binary and also manages the actual collection of inputs. These scripts are automatically generated before the honeypot is started and are using the Python API of GDB. This API allows us to fully automate the collection process of user inputs.

The debugging scripts sets a breakpoint on each of the instructions identified using the Firmware Analyzer as input sources. Then it disables pagination, which would otherwise stop the automated execution. Then the start address is recovered to account for Position Independent Executables. Finally before continuing the execution, the limit on the maximum number of elements that can be printed is disabled to allow us to retrieve the entire buffer. When the execution process hits such an instruction, the content of the registers identified using the Sources Parser module (e.g. r0) is read to retrieve the address of the buffer. Then the contents stored at this address are retrieved (0x123456) using the GDB instruction:

$$\text{x/s 0x123456} \tag{4.6}$$

Which returns the output:

$$\text{0x123456: "this is an input string"\n} \tag{4.7}$$

GDB then extracts the string for the output and creates an entry with this input, the location where it was discovered and the date time. This is then written to the output file.

The final job of the custom communicator is to open the FirmAE Debugger menu. This menu allows for custom introspection into the host system for the operator and can be used for the following tasks:

1. Start a socat session
2. Connect to a shell
3. Retrieve the list of active processes
4. Start a GDB server

5. Transfer a file to the host system
6. Stop the honeypot

4.2.2 Activity flow

In order to get a better understanding of the flow of the honeypot and the interactions between the different components, Figure 4.8 shows the activity flow of each component.

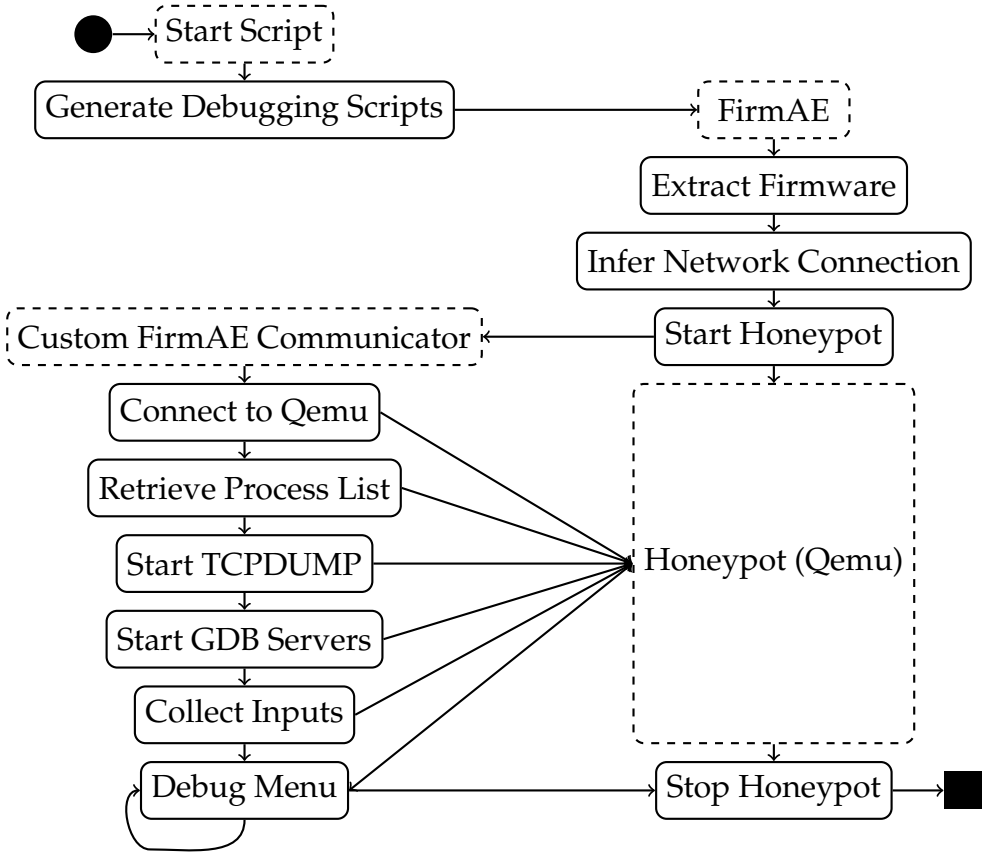

Figure 4.8: Overview of the activity flow in the honeypot.

The script starts with the output of the Firmware Analyzer and a firmware image and generates the debugging scripts which are user later on with GDB to control the collection of the user inputs on the honeypot. With these scripts generated, FirmAE is instantiated which extracts the firmware and prepares the custom kernel and the images to be executed by Qemu. FirmAE then tests if the firmware can be emulated successfully and automatically sets up the network connectivity.

When the setup phase is done, Qemu is started, which executes the honeypot. The honeypot opens the netcat session and at the same time, FirmAE starts the FirmAE communicator. The communicator connects to the netcat server on the honeypot and retrieves the list of processes that are active and starts a TCPDUMP session to capture all incoming network traffic. Based on this list and the list of symbolic links, the communicator sends a request for each process that requires debugging to the

honeypot to start a GDB Server. Processes are started on the host system to take the debugging scripts and connect to the GDB servers on the honeypot in order to collect the inputs remotely. The final step is to open the debug menu.

4.2.3 Custom Debugging Rules

Besides inputs from the Firmware Analyzer, there may be other already known vulnerabilities without vulnerability signatures available. To make it possible to detect attacks targeting these vulnerabilities, we added to option to add custom debugging rules and pass them to the honeypot via *.json* files. These rules allow to specify certain instruction addresses, registers to collect from and the maximum allowed size a buffer can contain. Furthermore, the start address of the vulnerable binary and a name for the custom rule are also provided.

Currently, custom debugging rules are limited to Buffer Overflow vulnerabilities. Extending this to other types of vulnerabilities is relatively easy. This can be done by adding more fields to the custom debugging rule and by adding code to the alert analyzer that verifies the input on the defined rules. The honeypot that was deployed to evaluate this system implemented a custom rule for a recently discovered vulnerability (more on this in Section 5).

4.2.4 Performance and limitations

The debugging process to collect inputs is based on GDB. Such an approach has its advantages and disadvantages. In this section, we want to discuss the performance and some of the limitations of the current prototype design of the honeypot.

First, debugging with GDB adds a significant overhead for each debugged process. This means that the number of input sources that we can collect inputs from is limited, since collecting inputs from all locations where user input is provided slows the honeypot down to an unusable state. For this reason, the honeypot only collects from input sources which are suspected to have vulnerabilities.

Secondly, GDB only has support to debug one process simultaneously. When a process that is debugged starts a child process, we either need to choose to continue debugging the child process and temporarily pause the parent process or continue debugging the parent process while letting the child processes run unimpeded. Since temporarily pausing the parent process is not an option in a honeypot environment, we are limited to the parent process. This is not necessarily a problem, since in all our tests on real firmware, the parent process was handling the user data. Debugging in all processes would need additional work and may require a different debugging framework. For this reason, we consider this as future work.

Finally, there is currently no way of starting a debugger on newly started processes. The debugger is only started on applications that are executed at system start. The debugging process is done remotely from the host system and therefore not suitable to quickly start the debugger on all new processes. For this reason, only the main services are debugged. This is also an improvement for a future version of this honeypot.

4.3 Input Analyzer

The final component of the prototype is used to detect if an input from the honeypot can actually reach a vulnerability and if it can trigger this vulnerability. This component is the Input Analyzer, which is explained in this section and is used to answer **Research Question 2**.

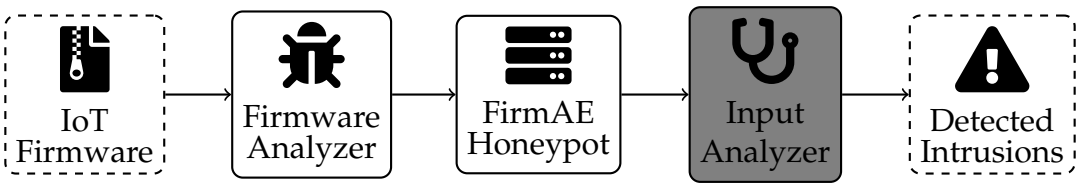

Figure 4.9: Schematic overview of the system components.

In order to answer **Research Question 2**, we need to discover how we can verify if the collected inputs from the honeypot can reach the vulnerability and we need to verify if the vulnerability can be triggered.

The Input Analyzer uses the output from the Firmware Analyzer, which includes the path constraints, the contents and information of the sink, the role and the symbolic tainted input variables. The information from the firmware analyzer is combined with the input sources collected by the Honeypot. This is done to reconstruct enough of the execution state to determine if an execution path can be reached with an input. Earlier, in the *Bug Finder* module, the *sink rule* was used to detect if a vulnerability was used. These sink rules can now be used to verify if the inputs can actually trigger the vulnerability.

In order to do so, a four step algorithm is used:

1. Reconstruct symbolic formula
2. Constrain the input sources
3. Verify satisfiability
4. Verify sink rule

To show how this algorithm is used, we use the code example that was used earlier in Section 4.1.5:

```
1 int network_copy(char network_input[]) {
2     dst_buffer = (char *) malloc(1000);
3     // test if the network_input starts with a data key
4     if memcmp(network_input, "http_referer", 12) {
5         strcpy(dst_buffer, network_input)
6     }
7     return dst_buffer
8 }
```

The vulnerable execution path can in this scenario be triggered if the network string starts with the data key *"http_referer"*. The *memcmp* would be detected as the role

function and would replace the network input variable with a symbolic variable: *BVS(taint_buf_1_2)*.

4.3.1 Reconstruct the symbolic formula

In order to know if this path can be reached with an input, we first need to know what constraints are used and reconstruct the symbolic formula from the symbolic execution. In this code example, there is only one constraint in the vulnerable path: the if statement which is based on the memcmp that verifies that the input starts with the data key. The generated path constraint is based on the output of the Simulated Procedure, as shown in Equation 4.8:

$$\begin{aligned} &\text{taint\_buf\_1\_2[0:7]} == \text{'h'} \ \wedge \text{taint\_buf\_1\_2[8:15]} == \text{'t'} \\ \wedge &\text{taint\_buf\_1\_2[16:23]} == \text{'t'} \wedge \text{taint\_buf\_1\_2[24:31]} == \text{'p'} \\ \wedge &\text{taint\_buf\_1\_2[32:39]} == \text{'\_'} \wedge \text{taint\_buf\_1\_2[40:47]} == \text{'r'} \\ \wedge &\text{taint\_buf\_1\_2[47:55]} == \text{'e'} \wedge \text{taint\_buf\_1\_2[56:63]} == \text{'f'} \\ \wedge &\text{taint\_buf\_1\_2[64:71]} == \text{'e'} \wedge \text{taint\_buf\_1\_2[72:79]} == \text{'r'} \\ \wedge &\text{taint\_buf\_1\_2[80:87]} == \text{'e'} \wedge \text{taint\_buf\_1\_2[88:95]} == \text{'r'} \end{aligned} \tag{4.8}$$

A path constraint is generated by the Firmware Analyzer for each possible branch in the Control Flow Graph of the analyzed function. This example has only one branch and therefore only one path constraint (Equation 4.8).

The first step of the algorithm is to take each of the path constraints and reconstruct the Symbolic Formula for this execution path. To reconstruct the symbolic formula, we can take each of the path constraints discovered and *chain* these constraints together using the $\wedge$ symbol.

In case taint is propagated during the symbolic execution, additional constraints generated by the Firmware Analyzer using the Summary Functions are chained together with all the path constraints to form the final symbolic formula. Finally, the Firmware analyzer also added extra constraints in case a symbolic variable can take a single value to speed up the analysis. These constraints are then chained to the symbolic formula as well.

Using this approach, the reconstructed symbolic Formula only consists of the Path Constraint in Equation 4.8.

4.3.2 Constrain the input sources

With a symbolic formula, we can verify if the path can be reached. Using the input collected from the Honeypot, the execution path can be analyzed and verified if the input can reach the vulnerable sink. Assume that the honeypot has collected an input from the input source, which are the arguments provided to the *memcmp* operation. The honeypot collected the following input from the first argument register *r*0:

http_referer: https://sidn.nl (4.9)

To constrain the symbolic formula, we need to apply another constraint to specify that the tainted input buffers can only be the collected input from the input. In this example, there is only one input variable, which means that only *taint_buf_1_2* needs to be constrained. When there are multiple variables, these constraints are applied to all the tainted buffers as a non-symbolic bit vector. Equation 4.10 shows the constraint added to the symbolic formula:

$$\text{taint\_buf\_1\_2} == \text{BVV}(\textit{“http\_referer: https://sidn.nl”}) \tag{4.10}$$

This constraint is then chained together with the full symbolic formula. The full symbolic formula that is used to verify reachability becomes (Equation 4.11):

$$\begin{aligned}
&\text{taint_buf_1_2[0:7]} == \text{‘h’} \wedge \text{taint_buf_1_2[8:15]} == \text{‘t’} \\
\wedge&\text{taint_buf_1_2[16:23]} == \text{‘t’} \wedge \text{taint_buf_1_2[24:31]} == \text{‘p’} \\
\wedge&\text{taint_buf_1_2[32:39]} == \text{‘_’} \wedge \text{taint_buf_1_2[40:47]} == \text{‘r’} \\
\wedge&\text{taint_buf_1_2[47:55]} == \text{‘e’} \wedge \text{taint_buf_1_2[56:63]} == \text{‘f’} \\
\wedge&\text{taint_buf_1_2[64:71]} == \text{‘e’} \wedge \text{taint_buf_1_2[72:79]} == \text{‘r’} \\
\wedge&\text{taint_buf_1_2[80:87]} == \text{‘e’} \wedge \text{taint_buf_1_2[88:95]} == \text{‘r’} \\
&\wedge \text{taint_buf_1_2} == \text{BVV}(\textit{“http_referer: https://sidn.nl”})
\end{aligned} \tag{4.11}$$

4.3.3 Verify satisfiability

The next step is to check if the Symbolic Formula is satisfiable. A constraint solver can be used on this formula to solve this. When this is satisfiable, we know that the execution path is reachable for this input.

In the project, the Z3 solver [60] is used and attempts to concretize possibly unknown variables and simplify the symbolic by replacing the symbolic variables with concrete values to solve the equation:

$$\begin{aligned}
&\text{‘h’} == \text{‘h’} \wedge \text{‘t’} == \text{‘t’} \wedge \text{‘t’} == \text{‘t’} \wedge \text{‘p’} == \text{‘p’} \wedge \text{‘_’} == \text{‘_’} \wedge \text{‘r’} == \text{‘r’} \\
&\wedge \text{‘e’} == \text{‘e’} \wedge \text{‘f’} == \text{‘f’} \wedge \text{‘e’} == \text{‘e’} \wedge \text{‘r’} == \text{‘r’} \wedge \text{‘e’} == \text{‘e’} \wedge \text{‘r’} == \text{‘r’} \\
&\wedge \textit{BVV}(\textit{“http_referer: https://sidn.nl”}) == \textit{BVV}(\textit{“http_referer: https://sidn.nl”})
\end{aligned} \tag{4.12}$$

As can be seen in Equation 4.12, the symbolic formula can be evaluated positively. This means that the input can reach the vulnerable sink. A string that does not start with “http_referer” would not be able to satisfy these rules.

4.3.4 Verify sink rules

With the reachability of a vulnerable path with an input of the honeypot verified, there is only one remaining step. This step is used to verify that a vulnerability can actually be triggered by the input.

This is done using *sink rules*. These are rules that define the way the vulnerability in a sink function or other type of vulnerability is triggered. Such a rule can be used to determine if a vulnerability can be triggered using an input collected by the honeypot and was also used by the Bug Finder module:

$$taint \wedge src > dst \tag{4.13}$$

Equation 4.13 shows the sink rule used to verify inputs for the *strcpy* method. In this example, the rule needs to verify that taint propagates into the sink function and that the size of the input in the *src* argument register is larger than the allocated size of the *dst* register.

The constraint solver may need to be used to discover the value that is stored in the *src* register, since this may be passed via a propagated taint. In this scenario, the symbolic formula is solved on the tainted variable in the *src* register and the size of this solution is used. Using this approach we can detect if an input can actually trigger the vulnerability, which is the final step in our prototype system.

4.4 Source code

The code of this prototype is published in three repositories. The first repository[2] contains all code related to the Firmware Analyzer and Alert Analyzer. The second repository[3] contains all code related to the honeypot. The third repository[4] contains several slight changes to FirmAE to allow it to be used in combination with the honeypot. This repository can be imported as a submodule in the honeypot repository.

All repositories contain a readme file with instructions to set up the project and all code is freely available under an open-source license.

[2] https://gitlab.sidnlabs.nl/christianscholten/iot-firmware-honeypot-static-analysis/

[3] https://gitlab.sidnlabs.nl/christianscholten/iot-firmware-honeypot

[4] https://gitlab.sidnlabs.nl/christianscholten/FirmAE/

Chapter 5

Deployment

The system as described in the previous chapter was deployed as a honeypot and used to collect inputs. In this chapter, we describe the way this system was set up in order to evaluate the system and its performance in Chapter 6.

5.1 Firmware requirements

Not all firmware images are suitable for testing and evaluation in this system. Some firmware images cannot emulate all network binaries. Others do have vulnerabilities discovered by the Firmware Analyzer, but not in emulatable binaries.

To accurately evaluate the system, the firmware image needs to fulfill the following requirements:

Requirement 1 *The system needs to be emulatable by* FirmAE *and must be reachable via the network.*

Requirement 2 *The system needs to be able to connect to the honeypot via the* FirmAE *communicator.*

Requirement 3 *The firmware needs to contain a verified vulnerability discovered by the Firmware Analyzer.*

Requirement 4 *The verified vulnerability needs to be in a binary that can be emulated by* FirmAE *and be reachable via the network.*

Requirement 5 *The system needs to be able to function as intended.*

These requirements are necessary to ensure proper functionality of the honeypot and an accurate way to measure the performance of the system. This does not mean all binaries are suitable for running the evaluation. Therefore, we need to test multiple firmware images to ensure these requirements are all met.

5.2 Selecting a firmware image

With these requirements in mind, several firmware images were tested. It quickly became clear that some firmwre images are not usable in the honeypot since they cannot be emulated or are unreachable in the honeypot or are unreachable over the network.

We selected the firmware from the FirmAE dataset [61]. The images of ASUS and TRENDNET in this dataset must be initialized manually. This process does not always work sufficiently: During tests with TRENDNET Wifi repeaters, the initialization process could not be finished due to missing (emulated) Wifi antennas.

Many other firmware images that could be emulated and were network reachable had different problems. In some cases, the web interfaces failed to execute.

Other binaries were able to start successfully with the web interface reachable, although for these binaries the FirmAE communicator failed to initialize. FirmAE opens a *netcat* connection by appending lines to the *preinit* of a binary. The problem is that the lines are added to the end of the file, which is not always reachable. This issue has been reported to the developers of FirmAE [62] and will be fixed in the next release of FirmAE.

Firmware images of routers by D-Link and TP-Link were most successful in emulating with full network connectivity and were most suitable for use as a honeypot. We chose two contenders that had known vulnerabilities: the D-Link DIR868L Revision B from 2016 (version 20161117) and the TP-Link C3200 V1 from 2015 (version 15083). Both of these images have vulnerabilities discovered by the test samples of Karonte.

The vulnerabilities in the D-Link image are in the *fileaccessd* binary and are not emulatable by the honeypot. This makes the images unsuitable for use in the honeypot. Instead, the TP-Link image does have a verified vulnerability in the updater module of the *httpd* binary, where one of the parameters is a 4096+ byte buffer. This buffer is copied into a buffer with a size of 556 bytes, with no constraints in place to limit the maximum size. This made the TP-Link image the most suitable for deployment in the honeypot.

The process of selecting a suitable image has shown that there are still large improvements to be made in the re-hosting process of firmware images and that the numbers provided by FirmAE in terms of successfully emulated and network reachable images are determined optimistically.

The main improvements that can be considered in FirmAE are:

1. Increase the number of binaries in a firmware image that can be emulated.
2. Solve the issue related to the initialization of the connection to the FirmAE Communicator / Debugger.
3. A solution needs to be discovered to better handle missing hardware.

FirmAE can keep running with missing hardware dependencies, but not all features in an image can be used with missing (emulated) hardware support of, for example, Wifi antennas.

The emulated services were exposed to the Internet. All discovered vulnerabilities are in the *httpd* service (port 80), so the debugging capabilities to collect inputs have been enabled for this service. Table 5.1 shows a list of the exposed services:

Name	Service	UDP / TCP	Debugging
21	ftp	TCP	No
22	ssh	TCP	No
23	telnet	TCP	No
80	http	TCP	Yes
1900	upnp	TCP	No
7547	cwmp	TCP	No
8200	trivnet1	TCP	No

Table 5.1: Exposed ports in the TP-Link Archer C3200 honeypot

5.2.1 Security considerations

When deploying honeypots security should be considered, since we do not want the honeypots to be used to launch attacks targeting normal devices. To prevent this, we followed the security considerations as discussed by Sokol and Andrejko [28] when deploying the honeypot.

The first measure taken is to reset the honeypot regularly to the original state. This means that attacks targeting the honeypot will not be long lived.

Secondly, the honeypot was deployed with network segmentation to ensure that it is not possible to connect from the honeypot to the host-system, while the host system is able to connect to it. Furthermore all debugging ports opened on the honeypot to control the honeypot and data collection are only opened to the host system and closed to the outside.

Finally, regular monitoring is necessary to ensure that the system keeps running correctly and to monitor for suspicious behaviour happening in the honeypot.

5.3 Modifications for TP-Link devices

There are some extra changes that were made that depended on the firmware image of TP-Link, which we want to discuss.

In the first version of the honeypot for the TP-Link firmware, the image was fully reset every 24 hours, so that every day the data collection could start with a fresh image. The problem that we discovered is that there is still a remaining problem in FirmAE which means that the emulation stops after approximately 90 minutes. To remedy this, the reset time has been updated to one hour. With this change this problem did not occur again.

Secondly, the TP-Link has some vendor specific quirks. One thing that appeared is that the web interface only accepts connections that have the following exact HTTP

header:

$$\textit{http-referer: http://192.168.0.1} \tag{5.1}$$

Connections to the web interface with a different hostname or IP address in this header are not accepted and are returned with a 401 error.

To allow connections, we used the tool *netsed* [63]. This tool allows to automatically rewrite incoming traffic on a port. We defined rules to rewrite this header to the *192.168.0.1* IP on the web interface on port 80.

5.4 Custom Debugging Rules

In order to evaluate the performance, we also added an extra Debugging Rule for a recently discovered vulnerability in the web interface of TP-Link devices (CVE-2021-29302 [64]).

This vulnerability is a buffer overflow in the *dm_checkString* method of the */lib/libcmm.so* library loaded by the *httpd* binary when the message is parsed. The size of the incoming message is not verified and can cause a buffer overflow by providing a large username or password in the password reset popup. This vulnerability could be used to cause the heap to overflow, which may be used to gain a remote shell [64]:

$$\begin{aligned}
&[/cgi/auth\#0,0,0,0,0,0\#0,0,0,0,0,0]0,3\\
&name = admin\\
&oldPwd = admin\\
&pwd = lys123aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\\
&aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
\end{aligned} \tag{5.2}$$

The function in which this vulnerability was identified is added as a custom rule to identify if this vulnerability is actually abused in real life. The deployed honeypot is set up to also gather information for this vulnerability. More details for this are discussed in the next chapter.

Chapter 6

Evaluation

In this chapter we evaluate the system as discussed in Chapter 4 which was deployed as discussed in Chapter 5. The evaluation in this chapter is performed in order to answer **Research Questions 3 & 4**, to evaluate the performance of our prototype and to evaluate the benefit of this solution compared to a regular IDS-based approach.

6.1 Collected inputs

The honeypot was deployed on May 12 2021 and started collecting all traffic for the vulnerability detected by Karonte. On May 18 2021 the custom debugging rule for CVE-2021-29302 [64] was deployed. All network traffic to the honeypot is collected in *.pcap* files. These files are analyzed in more detail in Appendix A.

The vulnerability detected by Karonte is in the updater module of the TP-Link firmware and is detected with the "http_rpm_update" data key. This vulnerability is confirmed to be a vulnerability using reverse engineering on the function, where we confirmed that this actually is a valid buffer overflow.

The other vulnerability, is more common and is due to the way that the HTTP body message is parsed on the web interface, where the size of the message is not parsed before being sent to the http daemon [64]. A proof-of-concept is provided that includes the code in which the crash occurs, which is the method that verifies all user input.

We were able to verify manually that we were able to collect inputs for CVE-2021-29302. Triggering the vulnerability detected by Karonte, on the other hand, is more complicated and we did not manage to manually trigger the vulnerable method via the honeypot.

On the honeypot, we did not collect any input data for either of these vulnerabilities. Since the collection process for CVE-2021-29302 includes all requests to the http interface, we know that there were no registered successful login attempts on the http server. The network traffic confirms that all traffic to port 80 are page loads that attempt to load malicious URLs and none of the *POST* requests to the interface are user interactions.

In comparison, the Telnet interface retrieved over fifteen thousand login attempts of which 213 were successful. The router was configured with the default username *admin* and the default password *admin*. Figure 6.1 shows an overview of the login attempts on the Telnet service showing the attempts and successful logins.

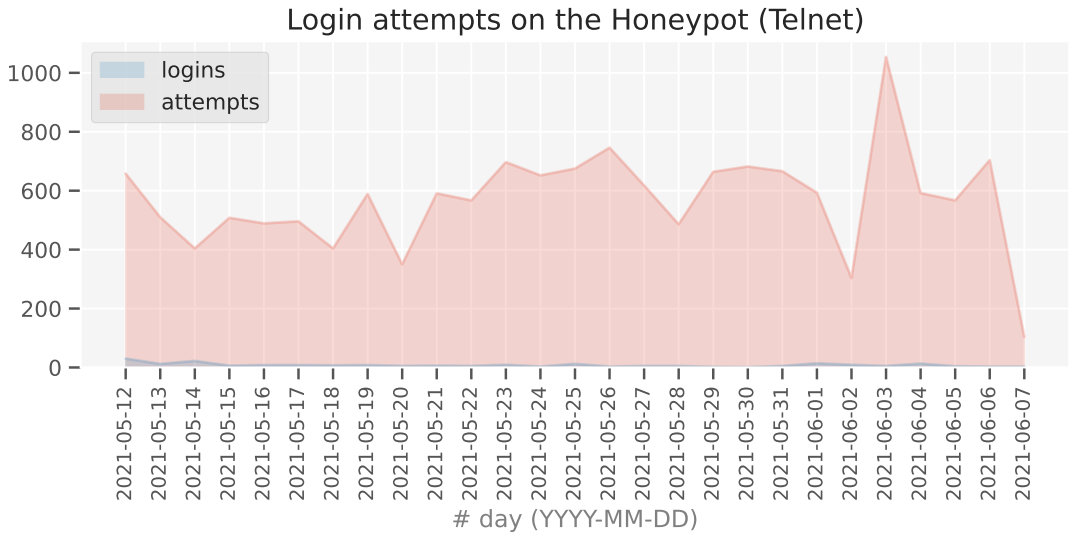

Figure 6.1: Login attempts on the honeypot (Telnet, port 23)

The data in this figure indicates that the honeypot is targeted on the Telnet port, with up to a thousand login attempts each day. Most of these attacks appear to be regular scans on the Telnet and SSH services from Mirai-related malware.

On a successful sign-in, each adversary attempts to gain a shell via either the *busybox*, *shell* or *system* commands, which are not available via this Telnet service. An example of such an interaction can be seen in Equation 6.1. The adversary then disconnect when it discovers that these commands are unavailable. There are no signs that any of these attempts were successfully exploiting the device.

$$\begin{aligned} &\text{TP-LINK(conf)\# /bin/busybox PEDO} \\ &\\ &\text{Command not found: /bin/busybox} \end{aligned} \tag{6.1}$$

This data shows that the honeypot is actively targeted, although not on the *http* service. Since we did not get any active attacks and user inputs on the web interface, we use an alternative way of evaluating the performance of this system. This means that we did not receive any successful login attempt on this server.

It is interesting that there is no active traffic targeting the *http* interface, since other research has shown that these custom manufacturer-based services are targeted for attacks as well. For example, the Mikrotik Winbox port is actively targeted for CVE-2018-14847 in order to set up PPTP tunnels [13].

A hypothesis may be that the known vulnerabilities in the *http* interface are not useful enough to warrant a custom attack targeting these devices. The vulnerabilities in the Winbox service, on the other hand, allow to read and write all files on disk and to discover the admin password. This vulnerability has the potential to cause remote code execution, with the proof-of-concept only causing a crash.

Targeting Telnet and SSH services with a brute-force attack is known to be successful, which is why it is commonly used by Mirai and other botnets [7]. A vulnerability that only targets one brand of routers is a lot less interesting than all Internet-of-Things devices configured with default or simple passwords on exposed services, which may be the reason why the honeypot did not receive any attacks.

6.2 Alternative Evaluation

The alternative evaluation is based on the *proof-of-concept* vulnerabilities for the custom debugging rules. For the vulnerability discovered by Karonte, we evaluate this using the decompiled code, since no proof-of-concept exists for this vulnerability. First, we evaluate the performance of the custom debugging rule.

6.2.1 Evaluating custom debugging rules

The *proof-of-concept* of CVE-2021-29302 is using the dialog to change the default password after initial setup as an example. If one of the parameters passed in this dialog is too long, then the receiving buffer overflows when verifying the input message. An example of this was shown in Equation 5.2. This is not the only place this vulnerability can occur, but for the evaluation we consider this input field to keep it consistent with the proof-of-concept. Since this input field can only be used once after initialization, the router requires a full reset in order to perform new tests at this input location.

During the tests, we discovered that the proof-of-concept initially did not work on our firmware image and returned the http *500* error (Internal Server Error). When the request is not authorized, it does not reach the vulnerable function and cannot trigger the vulnerability. To remedy this, we updated the proof-of-concept for this vulnerability, which can be found in Appendix B.1. Similar to the modifications made to the honeypot in order to get the TP-Link firmware running, this image also requires specific values in the request in order to be valid and requires authentication prior to allowing this command.

With the updated proof-of-concept, we managed to make the honeypot crash. The output from the log file containing the full crash can be found in Appendix B.2. Or when running the proof-of-concept in the *{iid}.final.serial.log* file in the *scratch* folder of FirmAE. After running the proof-of-concept, the *httpd* service stopped running, which stops the web service on port 80. This updated proof-of-concept works without the need to modify the firmware image and can therefore also be used under normal operation of the firmware.

When inspecting the crash log in Appendix B.2, there are some observations. First, it

shows that the crash occurred in the *dm_checkString* function, where it shows the the value is not valid. Then the application crashes with a segmentation fault (Signal 11). Furthermore, the dump of register contents show that multiple registers including the *pc*, *fp* and all argument registers now contain *0x62626262* as value. This corresponds to *bbbb* in ASCII, which is exactly the final half of the string passed by the *proof-of-concept* that was used to cause this crash, which shows that a well-crafted vulnerability may be used to jump to any line of code.

Running the *proof-of-concept* from Appendix B.1 on a running honeypot, it returns the following *json* entry:

```
{
    'binary': '/usr/bin/httpd',
    'function': 'dm_checkString',
    'basic_block': 'dm_checkString',
    'call_address': 'dm_checkString',
    'buffer_reg': ('r1',),
    'buffer_contents': ['
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
    '],
    'date_time': '2021/06/19 14:41:53'
 }
```

This entry shows that we collected inputs from the *dm_checkString* method in the first basic block of this method and collected the input from the *r1* register using the GDB server running on the *httpd* binary. The date time then shows that the input is collected on *June 19 2021*.

The next step in the evaluation is to verify the inputs using the Alert Analyzer. The information provided by the *Custom Rule* shows the maximum size of the buffer, which can be used to determine the inputs.

The Input Analyzer algorithm is easier for Custom Rules, since this is not based on a symbolic path formula. Instead, the input was retrieved directly from the buffer, which means that only the size of the input needs to be verified, as shown by the output of the Alert Analyzer for the input collected by the Honeypot using the *proof-of-concept*:

```
Testing CVE-2021-29302 with input:
        ['aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
This input triggers a vulnerability: True
```

Similarly, a smaller input can be verified that is does not trigger a buffer overflow:

```
Testing CVE-2021-29302 with input: ['admin']
This input triggers a vulnerability: False
```

Performance metrics The advantage of using debugging rules is that the inputs can be collected directly at the location of the vulnerable function. The custom rule used in this example specified the Assembler instruction at which the overflow occurs, which makes it easy to capture the exact input.

In order to accurately detect a buffer overflow, we need to know two factors:

1. The exact Assembler instruction in which the vulnerability occurs
2. The exact allocated buffer size in Assembler

With these two factors, we can determine when a vulnerability is triggered. This does not mean that an attack is successful. The payload from the *proof-of-concept*, for example, only manages to crash the system when it overrides the register contents. Smaller inputs may not have the same outcome, even though the buffer did overflow.

Since we only consider buffer overflows in this research, we evaluate the custom debugging rules only for this type of vulnerability. The system can detect the vulnerabilities with few false positives and false negatives, since the input was recorded directly at the vulnerable location.

There may still be some factors that may decrease accuracy. For example, the allowed buffer size may be slightly larger than uncompiled C code, since compilers may require each new byte to be aligned at addresses that are divisible by the *word* size [65]. This can be avoided by determining the allocated space based on the machine code, instead of the uncompiled code. This is only applicable to inputs close to the size of the allocated buffer. Larger inputs were always evaluated accurately in our prototype.

Another aspect to consider is that the inputs collected may not always be user-provided inputs. The algorithm used in the Binary Dependency Graph to detect roles can detect input sources that are likely to be user inputs, but this is not necessarily the case. In the D-Link DIR-868 firmware, for example, we detected a potential user input that in practice only had *$get* or *$post* as possible values. Furthermore, the input location used in the Custom Rule for CVE-2021-29302 is in a method that was used to verify the validity of input strings. This vulnerable method may not necessarily only be used to handle user data. In such a case there may be incoming noise, which means that more data needs to be analyzed.

Comparison to IDS-based signatures Not only the accuracy and precision of the detection system are important, but also how this system compares to other approaches, such as IDS-based solutions.

An IDS-based solution has two factors to consider when comparing both solutions:

1. The availability of IDS-signatures
2. The quality of IDS-signatures

First, the availability of signatures. There are large sets of IDS-signatures available, but still many vulnerabilities do not have an IDS-signature. The vulnerability used in this evaluation does not have a signature available, even though it has a *High (8.1)* severity on the CVSS scale [64].

When a signature is available, the quality also needs to be considered. The signature should have a high accuracy and try to minimize False Positive and False Negative alerts. The vulnerability can be used in nearly any input field, which means that the

signature needs to encompass all of these different variations and input locations. This consideration makes it significantly more difficult to create a signature with a good accuracy and precision.

As an example, the following payload in the field that sets up the SSID for the 2.4GHz WiFi network also manages to crash the web service:

```
[LAN_WLAN#1,1,0,0,0,0#0,0,0,0,0,0]0,5
X_TP_PreSharedKey=12345670
smartConnect=0
enable=1
SSID=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
SSIDAdvertisementEnabled=1
```
(6.2)

The solution proposed in this paper does not have this problem, since it detects the payload in the vulnerable function itself and therefore does not need a signature for each different variation of this payload, which makes detection more effective for this type of vulnerabilities. Since we can detect the payload at all input locations, our system has a better detection rate of True Positives. An IDS-based solution has more False Negatives due to the larger variation in possible exploits for the same vulnerability.

6.2.2 Evaluating the Firmware Analyzer

In the previous subsection, we evaluated the use of *Custom Debugging Rules* and how well the detection of vulnerabilities worked. Although we did not capture any attack exploiting a vulnerability, we showed that the custom debugging rules are a useful way of detecting intrusions. Now we want to perform a similar evaluation for the vulnerabilities detected by the *Firmware Analyzer*.

We did not find a way to trigger the vulnerability in the official way using the web interface. With some careful inspection of the method, it appears that the Control Flow Graph did not find any predecessor block to this method, which could mean that either Angr failed to detect a data flow to this function, or that this method is unreachable. This is unfortunate, since this binary is one of the few binaries that worked well in combination with FirmAE and the Firmware Analyzer. This means that this vulnerability cannot be triggered under normal operation of the firmware.

This does not mean that we cannot evaluate this system. Since we have shown using the previous subsection that we can correctly collect inputs from the detected input locations, we can use automatically generated input entries to evaluate the Alert Analyzer for this vulnerability. One of the aspects we consider is the evaluation time to verify an input.

To start the evaluation, we generated a thousand inputs of increasing length using the *fake_input_generator.py* file in the static analysis repository. These inputs are then used

as inputs for the Alert Analyzer. The fake inputs used for this evaluation are stored in the repository[1].

There are two factors to consider: *reachable* and *trigger*. Reachable means that the input from the collected location can reach a sink, while trigger means that the input can trigger the vulnerability. There are no branches in the path from the input location to the sink, which means that all inputs can reach the sink. This means that no values should be classified as *unreachable*. The buffer has a size of 4144 bits (518 bytes) and only inputs larger or equal to this size return *True* for the trigger. Since all inputs for this vulnerability should be classified as *reachable*. This factor is evaluated later in this chapter using a more representative sample.

After running the evaluation, the alert analyzer returned the following results (Table 6.1):

	trigger	
	T	*F*
expected	483	517
actual	483	517

Table 6.1: Expected and actual classification of the inputs

As we can see, all inputs are classified correctly. This process of testing all inputs does take a significant amount of time (3 hours, 13 minutes and 15 seconds to evaluate all inputs). This is on average 11.60 seconds per input. The test on reachability is memory intensive, since it needs to calculate if the entire formula can be satisfied.

Testing reachability Since there is only a single path from input to the vulnerable function, all inputs should be classified as reachable using the vulnerability discovered by KARONTE. This makes it impossible to properly evaluate this feature of the Alert Analyzer. Therefore, we evaluate this using a different approach.

When creating the Firmware Analyzer, we first used a custom binary that we created with a path to a vulnerable function. This example binary can be found in Appendix C.1 and is a simple binary that follows the principles used by the Firmware Analyzer (Section 4.1).

The user input is provided using the *scanf* method and detected with the *http* data key in the *memcmp* function. Then the *memcmp* verifies if the string starts with http and if not, it enters the *else*-condition with the buffer overflow. The full path constraint in the style used by ANGR for this path can be found in Appendix C.2. The *taint_buf_rdi__0_64* variable is the variable that we applied taint to.

The symbolic formula evaluates to true if the condition inside does not evaluate to **0x0**. We can confirm that the only value that does not enter this path is the string

[1] `https://gitlab.sidnlabs.nl/christianscholten/iot-firmware-honeypot-static-analysis/-/tree/master/collected_inputs/tp-link/Archer_C3200_V1_150831.zip/Archer_C3200_V1_150831.zip_fake_inputs.pk`

"http". First, we invert the final part of the constraint from the *memcmp* from **... != 0x0** (Appendix C.1 Line 33) to **... == 0x0**. This symbolic formula is then provided to the solver which solves the input variable for the first four bytes of the tainted buffer and ask for up to 10 solutions. The solver then only returns a single value:

$$Out[21] : [1886680168] \tag{6.3}$$

This number evaluates to the string "ptth", which is "http" stored in a little-endian manner. Since we only receive one solution, we can be certain that this symbolic formula is correct for this code sample.

Performance metrics The example above assumes a perfect scenario where all variables are known. There might be scenarios where the path depends on a variable that was provided earlier than the user input. In this case we do not have a way to know that variable. This increases both the number of possible solutions that evaluate to true and the False Positive rate. This is not necessarily a problem, since there is also a second check that verifies the *sink rule* for the vulnerability, which filters out many of these False Positives. Due to the quality of the Simulated Procedures on which the symbolic formula is based and the fact that the inputs are collected directly during execution, the rate of False Negatives is low. In our tests evaluating the reachability, we did not manage to get a False Negative for any of the inputs.

In conclusion, we can accurately predict if an input can trigger a vulnerability. The reachability of a vulnerable function may be based on earlier data, which may not be provided by the user. This could increase the rate of False Positives and make it easier to satisfy the full condition. Each alert that indicates that a vulnerability can be reached and triggered still needs some manual verification to confirm that the input is actually an attack.

Comparison to IDS-based solutions Since we do not have a signature that we can compare this vulnerability to, it is difficult to directly compare the effectiveness of the vulnerabilities detected by the Firmware Analyzer with an IDS-based signature. We can still reason about the differences in performance.

From the previous paragraph on Accuracy and Precision, we know that the chance of False Negatives is low, with a higher chance of getting classified as False Positives, due to variables in the symbolic formula that may not be initialized. Depending on the quality of the signature, an IDS-based solution can have a bias to either one of them.

The approach proposed in this thesis is not necessarily a replacement of an IDS. The best use case for this detection system in combination with the *Custom Debugging Rules* is for vulnerabilities with no signature, since we have shown that this solution is capable of accurately predicting if an input can actually trigger a buffer overflow. In future work we may need to evaluate this honeypot for different types of vulnerabilities to evaluate if this can effectively be used for other types of vulnerabilities as well.

The direct integration with the Firmware Analyzer is less of a success. Only some of the vulnerabilities detected can be triggered manually and FirmAE only supports emulation of a subset of all firmware images. We already had difficulty finding a suitable firmware image where the Firmware Analyzer could find a vulnerability and could be emulated using FirmAE. Still, FirmAE is in active development which might make this less of a problem in the future.

We do think that the *Custom Debugging Rules* provide a better way of adding input sources for the honeypot by basing the input sources on already known vulnerabilities. For this reason, we recommend any future work to prioritize the Custom Debugging Rules over the integration of the Firmware Analyzer.

Chapter 7

Conclusion

In this thesis, we attempted to answer the question on how effective static analysis techniques are in automating the detection of zero-day attacks in high-interaction IoT honeypots. To answer the main question, four research questions were defined, which should aid in answering the main question. The first two research questions are answered in Section 4.1 and are answered using a practical approach using a prototype consisting of three components.

First, the *Firmware Analyzer* is a component based on Karonte [9] that uses static analysis techniques to discover vulnerable execution paths in an IoT firmware image. For these paths, we store all constraints in the path from the input to the vulnerable function and all other necessary information to reconstruct the vulnerable path at a later stage.

The second component is a high-interaction honeypot that uses the firmware re-hosting tool FirmAE [8] to run the firmware image outside of the intended device. This honeypot also used debugging tools to collect user inputs at the vulnerable input locations discovered by the Firmware Analyzer.

The final component, the *Input Analyzer*, takes the inputs collected by the honeypot, the path constraints and all other parameters from the output of the Firmware Analyzer and reconstructs the vulnerable execution path in order to discover if the input by the honeypot can actually reach and trigger the vulnerability.

This prototype was then deployed as a honeypot for a firmware image of the Archer C3200 router by TP-Link. Using the information collected by the honeypot and our own evaluation based on proof-of-concept attacks we were able to evaluate this system in Section 6 and answered Research Questions 3 and 4 based on this evaluation.

The answer to **RQ1**, *How are the input sources of an IoT honeypot chosen to drive the static analysis?*, is that the input sources can be detected using *Communication Paradigm Finders*. These modules detect likely locations of user input and how these inputs are propagated through the binary. The communication between binaries is detected using known communication patterns, where each binary is given a role: *Setter* or *Getter*. To detect that user input is used in this we use *data keys*. These are keywords related to

network or user data. For each location where a role is detected, we store the type or role and information of this location. The roles for which a vulnerable path exists are the ones used as input source in the honeypot.

Similarly, to answer **RQ2**, *What steps need to be taken to determine if an input can reach and trigger a vulnerability?*, it was found that we can recreate the execution path of a potentially vulnerable input using a four step algorithm. First, we need to recreate the symbolic formula using the path constraints that the Firmware Analyzer identified. The next step is to constrain the formula by constraining the symbolic variable for the input with the input variable from the honeypot. To detect if the input can reach a vulnerable function, we need to verify that the formula can be solved using a constraint solver. The final step is to verify that the input can trigger the vulnerability using *sink rules*.

Additionally, to answer **RQ3**, *How accurate are the alerts triggered by the static analysis using performance metrics?*, we used the honeypot data and proof-of-concept attacks. We discovered that the system can detect attacks with few False Negatives. False Positives are possible in case the symbolic formula depends on variables outside of the collected inputs, although these False Positives are limited due to the additional check using the *sink rules*. Nevertheless, the system is more likely to return False Positives than False Negatives.

The answer to the final question, **RQ4**, *How does the performance of the static analysis compare to an IDS-based approach?*, is more nuanced. Our system is not necessarily a replacement for IDS-based signatures, but can be a useful addition in case signatures do not exist. For this reason, we added the option to add *Custom Debugging Rules* that can detect attacks targeting existing vulnerabilities. These custom debugging rules are where our system works the best, since we can easily add rules for vulnerabilities with a high score on the CVSS scale for which no signature exists. We showed this by evaluating Custom Debugging Rules for CVE-2021-29302 [64]. The vulnerabilities discovered by the firmware analyzer, in comparison, need to be verified to ensure that they are actually vulnerabilities that can be triggered. This also needs to ensure that the image can be emulated using the honeypot. As such, this only leaves a small fraction of images that can be used, which in combination with the long run times of the Firmware Analyzer limits the usefulness of this solution.

With the research questions answered, we can answer the main question. We discovered from **RQ3** that we can detect attacks with a good accuracy. There are some factors that limit the effectiveness of the static analysis. Although FirmAE can emulate more firmware images compared to its predecessors, the emulation rate is still insufficient. We were only able to find a single firmware image that had a vulnerability that was detected by the Firmware Analyzer which could be emulated using FirmAE in the honeypot. The addition of custom debugging rules, on the other hand, does show potential. We showed that we can accurately detect attacks targeting a buffer overflow vulnerability for which no signature exists using these rules. The custom rules are only limited by the emulation rate of FirmAE, which makes it easier to scale these rules to more devices compared to the vulnerabilities discovered by the Firmware Analyzer.

In conclusion, we do not think that using static analysis techniques for IoT firmware that is then emulated does not provide sufficient benefits compared to IDS-based solutions. The limitations in the emulation rate of firmware re-hosting tools. There are not enough firmware images that can benefit from this solution due to these limitations. This may change in the future when future versions of firmware re-hosting tools can emulate a larger set of images.

7.1 Future Work

This prototype only considers buffer overflow vulnerabilities. The prototype could be extended to other types of vulnerabilities, such as *use-after-free* vulnerabilities.

Secondly, we demonstrated that the *Custom Debugging Rules* do show potential. The requirement that our prototype should also have a vulnerability detected by the static analysis limited the scale on which we could analyze this feature. Our solution showed that combining static analysis with the honeypot does not provide enough benefit, due to the limitations in the emulation rate and the aforementioned requirement. New research could be done that evaluates the effectiveness of the Custom Debugging Rules on a larger scale without the use of static analysis techniques.

Finally, more work needs to be done on the re-hosting problem of firmware images. There are still too many images that cannot be emulated fully. When a new solution is created that can emulate more firmware images, then it may be of value to re-evaluate our conclusion to this research.

Bibliography

[1] Gartner. Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020, August 2019. URL https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io. (accessed: 15.01.2021).

[2] Unit 42. 2020 Unit 42 IoT Threat Report. Technical report, Palo Alto Networks, January 2020. URL https://iotbusinessnews.com/download/white-papers/UNIT42-IoT-Threat-Report.pdf. (accessed: 10.12.2020).

[3] Forum of Incident Response and Security Teams, Inc. Common Vulnerability Scoring System, 2021. URL https://www.first.org/cvss/. (accessed: 20.07.2021).

[4] Mario Ballano Barcena and Candid Wueest. Insecurity in the Internet of Things. *Security response, Symantec*, 2015.

[5] Shodan, 2021. URL https://shodan.io. (accessed: 11.12.2020).

[6] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. ZMap: Fast Internet-wide scanning and its security applications. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 605–620, 2013.

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai Botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[8] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *Annual Computer Security Applications Conference*, pages 733–745, 2020.

[9] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

[10] Cristian Hesselman, Jelte Jansen, Marco Davids, and R de O Schmidt. SPIN: a user-centric security extension for in-home networks. Technical Report SIDN-TR-2017-002, SIDN Labs, July 2017.

[11] Elmer Lastdrager, Cristian Hesselman, Jelte Jansen, and Marco Davids. Protecting Home Networks From Insecure IoT Devices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6. IEEE, 2020.

[12] Juan David Guarnizo, Amit Tambe, Suman Sankar Bhunia, Martín Ochoa, Nils Ole Tippenhauer, Asaf Shabtai, and Yuval Elovici. Siphon: Towards scalable high-interaction physical honeypots. In *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*, pages 57–68, 2017.

[13] João M Ceron, Christian Scholten, Aiko Pras, and Jair Santanna. MikroTik Devices Landscape, Realistic Honeypots, and Automated Attack Classification. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2020.

[14] Ronny Ko and James Mickens. DeadBolt: Securing iot deployments. In *Proceedings of the Applied Networking Research Workshop*, pages 50–57, 2018.

[15] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27, 2006.

[16] Philipp Morgner and Zinaida Benenson. Exploring Security Economics in IoT Standardization Efforts. *CoRR*, abs/1810.12035, 2018.

[17] Xueying Yang, Evgenios Karampatzakis, Christian Doerr, and Fernando Kuipers. Security vulnerabilities in LoRaWAN. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 129–140. IEEE, 2018.

[18] Cesar Cerrudo, Esteban Martinez Fayo, and Matias Sequeira. LoRaWAN Networks Susceptible to Hacking: Common Cyber Security Problems, How to Detect and Prevent Them. Technical report, IOActive, 2020. URL `https://act-on.ioactive.com/acton/attachment/34793/f-87b45f5f-f181-44fc-82a8-8e53c501dc4e/1/-/-/-/-/LoRaWAN%20Networks%20Susceptible%20to%20Hacking.pdf`. (accessed: 28.01.2020).

[19] Qualcomm Technologies Inc. Qualcomm and Google Announce Collaboration to Extend Android OS Support and Simplify Upgrades, December 2020. URL `https://www.qualcomm.com/news/releases/2020/12/16/qualcomm-and-google-announce-collaboration-extend-android-os-support-and`.

[20] ETSI. Cyber Security for Consumer Internet of Things: Baseline Requirements. Standard ETSI EN 303 645, European Telecommunications Standards Institute, Sophia Antipolis, Fr, June 2020. URL `https://www.etsi.org/deliver/etsi_en/303600_303699/303645/02.01.01_60/en_303645v020101p.pdf`.

[21] California Legislative Information. SB-327 Information privacy: connected devices., 2018. URL `https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201720180SB327`. (accessed: 20.01.2021).

[22] Wenjun Fan, Zhihui Du, David Fernández, and Victor A Villagra. Enabling an anatomic view to investigate honeypot systems: A survey. *IEEE Systems Journal*, 12(4):3906–3919, 2017.

[23] Iyatiti Mokube and Michele Adams. Honeypots: concepts, approaches, and challenges. In *Proceedings of the 45th annual southeast regional conference*, pages 321–326, 2007.

[24] Niels Provos. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2, page 4, 2003.

[25] Ubiquity USG/UDM Honeypot, 2020. URL `https://help.ui.com/hc/en-us/articles/360006893234-UniFi-USG-UDM-Configuring-Internet-Security-Settings#7.2`. (accessed: 15.12.2020).

[26] Tongbo Luo, Zhaoyan Xu, Xing Jin, Yanhui Jia, and Xin Ouyang. Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices. *Black Hat*, 2017.

[27] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. IoTPOT: analysing the rise of IoT compromises. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[28] Pavol Sokol and Maroš Andrejko. Deploying honeypots and honeynets: Issues of liability. In *International Conference on Computer Networks*, pages 92–101. Springer, 2015.

[29] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 135–150, 2019.

[30] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[32] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–11, 2015.

[33] Eric David Gustafson. *Discovery and Remediation of Vulnerabilities in Monolithic IoT Firmware*. PhD thesis, UC Santa Barbara, 2020.

[34] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS*, volume 16, pages 1–16, 2016.

[35] binwalk, 2021. URL `https://www.refirmlabs.com/binwalk/`. (accessed: 14.12.2020).

[36] Alexander Vetterl and Richard Clayton. Honware: A virtual honeypot framework for capturing CPE and IoT zero days. In *2019 APWG Symposium on Electronic Crime Research (eCrime)*, pages 1–13. IEEE, 2019.

[37] Honware, 2019. URL `https://honware.org`. (accessed: 14.12.2020).

[38] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340, 2014.

[39] Karl Koscher, Tadayoshi Kohno, and David Molnar. {SURROGATES}: Enabling near-real-time dynamic analyses of embedded systems. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[40] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *NDSS*, volume 14, pages 1–16, 2014.

[41] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, volume 18, pages 1–11, 2018.

[42] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[43] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Sec)*, pages 1–18, 2020.

[44] Alexander Vetterl and Richard Clayton. Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[45] Michael Brengel, Michael Backes, and Christian Rossow. Detecting hardware-assisted virtualization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 207–227. Springer, 2016.

[46] Alexander Kedrowitsch, Danfeng Yao, Gang Wang, and Kirk Cameron. A first look: Using linux containers for deceptive honeypots. In *Proceedings of the 2017 Workshop on Automated Decision Making for Active Cyber Defense*, pages 15–22, 2017.

[47] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility Is Not Transparency: VMM Detection Myths and Realities. In *HotOS*, 2007.

[48] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[49] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[50] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. DECAF++: elastic whole-system dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 31–45, 2019.

[51] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 569–584, 2012.

[52] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. DTaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[53] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[54] Javad Yousefi, Yasser Sedaghat, and Mohammadreza Rezaee. Masking wrong-successor Control Flow Errors employing data redundancy. In *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 201–205. IEEE, 2015.

[55] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[56] OpenBSD. strlcpy(3) - Linux man page, April 2013. URL `https://linux.die.net/man/3/strlcpy`. (accessed: 20.01.2021).

[57] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[58] Jeff Hiner. We Rewrote Our IoT Platform in Rust and Got Away With It, July 2019. URL `https://medium.com/dwelo-r-d/we-rewrote-our-iot-platform-in-rust-and-got-away-with-it-2c8867c61b67`. (accessed: 20.01.2021).

[59] Red Hat Inc. Position Independent Executables (PIE), November 2012. URL `https://access.redhat.com/blogs/766093/posts/1975793`. (accessed: 25.05.2021).

[60] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[61] pr0v3rbs. FirmAE dataset, 2021. URL `https://drive.google.com/file/d/1hdm75NVKBvs-eVH9rKb5xfgryNSnsg_8/view?usp=sharing`. (accessed: 13.07.2021).

[62] Can't connect to shell, FirmAE Issue #19, May 2021. URL `https://github.com/pr0v3rbs/FirmAE/issues/19`. (accessed: 03.06.2021).

[63] Mats Erik Andersson. NETSED man page, 2010. URL `http://manpages.ubuntu.com/manpages/trusty/man1/netsed.1.html`.

[64] liyansong2018. CVE-2021-29302, 2021. URL `https://github.com/liyansong2018/CVE/tree/main/2021/CVE-2021-29302`. (accessed: 03.06.2021).

[65] EventHelix. https://www.eventhelix.com/embedded/byte-alignment-and-ordering/, 2021. URL `https://www.eventhelix.com/embedded/byte-alignment-and-ordering/`. (accessed: 22.06.2021).

[66] ColdHeat. Geoacumen-Country Database, 2021. URL `https://github.com/geoacumen/geoacumen-country`. (accessed: 09.06.2021).

[67] The Spamhaus Project. The World's Worst Spam Enabling Countries, 2021. URL `https://www.spamhaus.org/statistics/countries/`. (accessed: 09.06.2021).

[68] The Spamhaus Project. The World's Worst Botnet ASNs, 2021. URL `https://www.spamhaus.org/statistics/botnet-asn/`. (accessed: 10.06.2021).

Appendix A

Evaluation of the honeypot data

We evaluate the raw data collected by the honeypot in order to see certain trends on the attackers targeting the honeypot. This analysis evaluates the ports that are targeted most, the countries the attackers originate from, the Autonomous Systems (AS) and several other statistics.

A.1 General honeypot statistics

The honeypot was deployed on May 12, 2021 which collected the network traffic as *.pcap* files and it was set up to collect inputs for the vulnerability discovered by the Firmware Analyzer. On May 18 2021, the custom debugging rules for *CVE-2021-29302* were added to start collecting inputs for this vulnerability as well. Figure A.1 show the traffic to the honeypot for each day in terms of packets, flows and connections. The number of flows and connections only differs slightly, which explains why only two colours are identifiable in this figure.

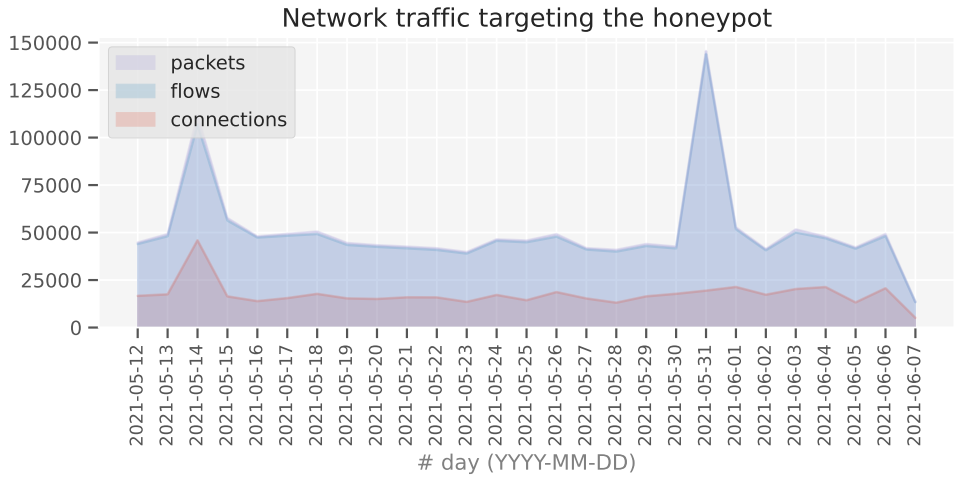

Figure A.1: Network traffic targeting the TP-Link honeypot

The spikes on May 14 and May 31 are caused by brute-force attacks on the *ssh* service

in combination with full port scans from several hosts at the same time. The default telnet service is set up in such a way that it only allows five attempts at a time, after that it requires a wait of 600 seconds. This approach appeared to stop many larger brute-force attacks. The traffic on the final day is lower due to the honeypot stopping halfway during the day.

Destination port In Table 5.1 we shows the exposed services of the honeypot. In order to see if the attackers prefer to attack one service over the other, we made a graph showing the top ports that were connected to. Figure A.2 shows the traffic to the top ten ports, including closed ports. Similarly, Figure A.3 shows the total number of network flows received on all exposed ports.

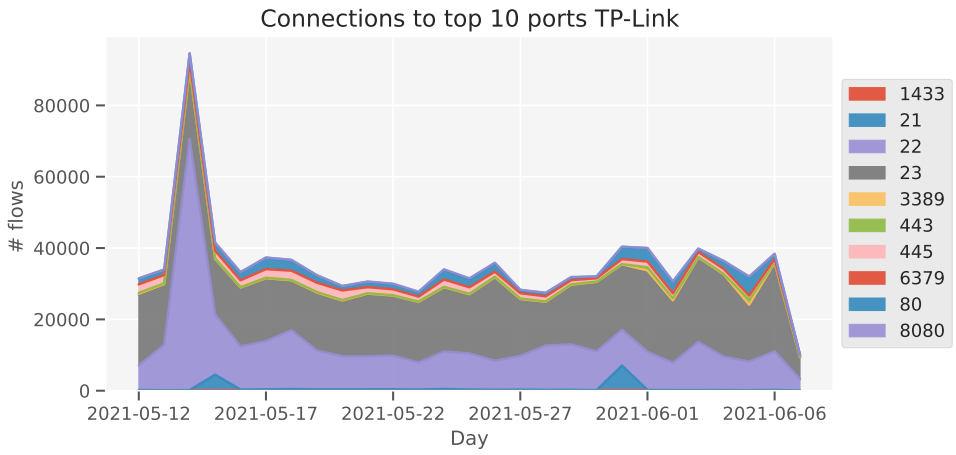

Figure A.2: Network traffic to the top 10 ports

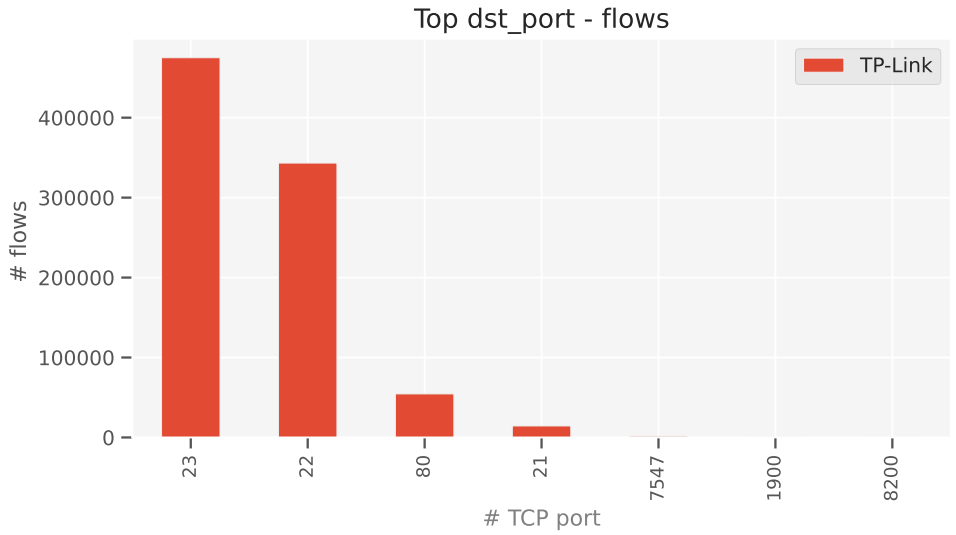

Figure A.3: Networks flows to the exposed ports

The most common services reached are the *telnet* and *ssh* services on port 23 and 22. The web service on port 80 only gets a fraction of traffic, while the *ftp* service on port 21 only

gets traffic in bursts on certain days. All the other top ports are closed. All network traffic targeting these ports are from either network scans or attackers attempting to connect to the *smb* (port 443) or *https* services (port 445).

Another interesting observation is that the peak of network traffic on May 31 is not visible in Figure A.2. This is because the burst of traffic was due to several full network scans on the entire port range, which means that the traffic is distributed over all ports. In contrast, the attack on May 14 is mostly traffic over *ssh*, which does show up on the graph.

A.2 Country of origin

Another statistic to look at are the countries where the attacks originate. Using the Geoacumen-Country database [66] we mapped the IP-addresses to the corresponding country. Figure A.4 shows an overview of the countries from which we received most network flows.

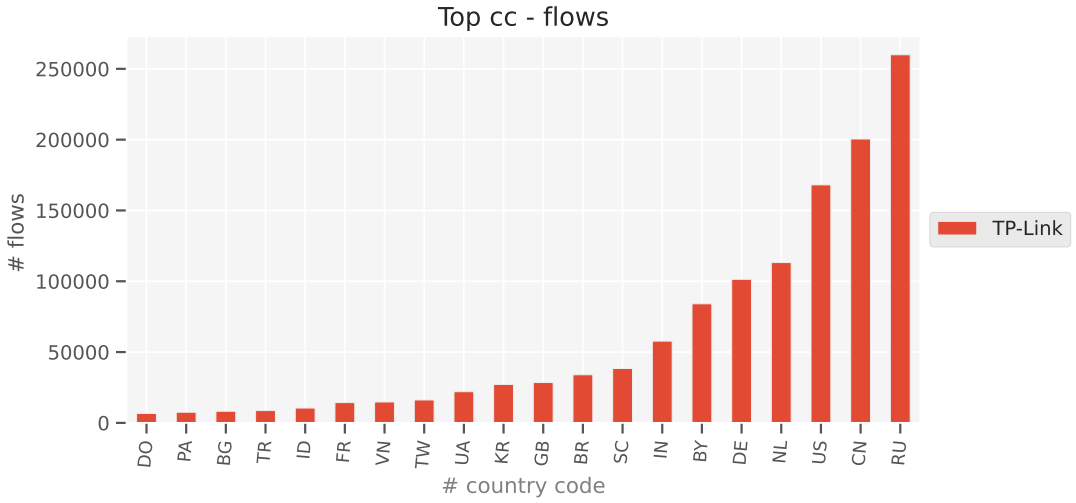

Figure A.4: Top countries in terms of network flows

There is a large overlap in the top countries compared the Spamhaus list of top countries enabling spam [67], with all of the top countries in the Spamhaus project except Japan (JP) and Hong Kong) appearing in nearly the same order as this list. There are some interesting countries in the top list. Especially Belarus (BY) and The Seychelles (SC) are interesting entries with a high number of flows originating from those countries.

In terms of total connections, there are a few differences. First, China is now the top country instead of Russia and now all countries from the Spamhaus list appear in the graph, except Japan. The Seychelles does not appear in this list anymore, which indicates that it has been fewer larger attacks originating from there.

When looking at the top countries, the top fifteen countries provide 87.47% of the total number of network flows. Figure A.5 shows this relation as a cumulative distribution function (CDF) of the network flows and a probability density function (PDF) that shows the percentage of flows for each country code.

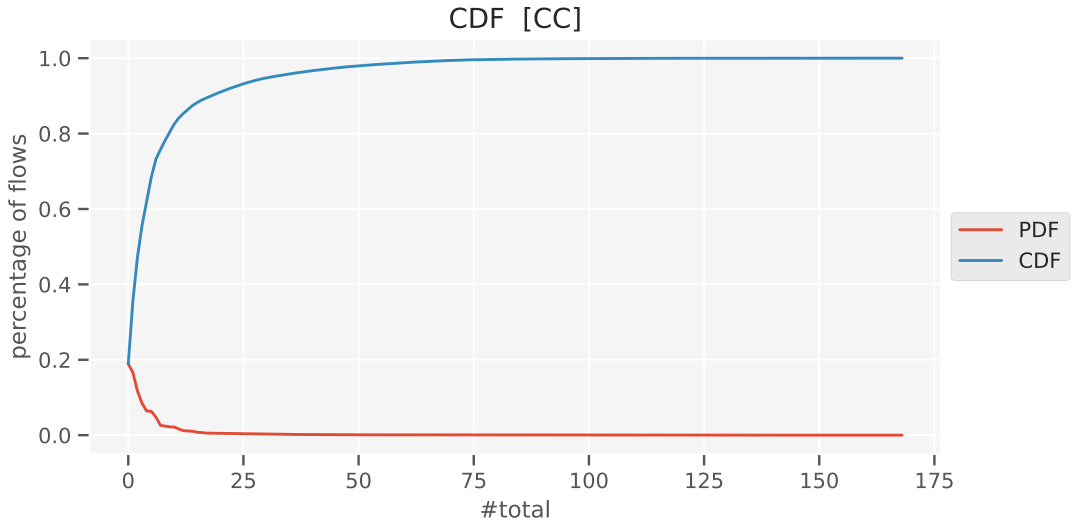

Figure A.5: Cumulative distribution function of the Country Codes targeting the honeypot

A.2.1 Top Autonomous Systems (AS)

The Autonomous Systems the flows belong to is another aspect to consider. Some networks may be more interesting to host botnets on due to the countries technical policies and socioeconomic factors according to the Spamhaus Project [68]. Figure A.6 shows the top Autonomous Systems that we received traffic from in terms of network flows and the number of unique IP addresses.

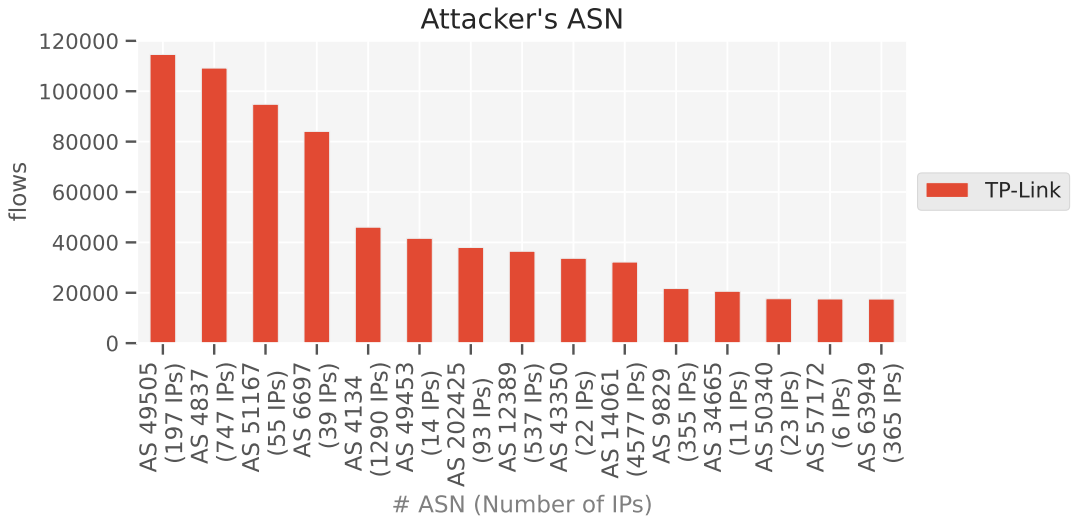

Figure A.6: Top Autonomous Systems in terms of network flows and unique IPs

By comparing this to the top list of Autonomous Systems infected with botnets from the Spamhaus Project, we can identify AS4134 (China Telecom) and AS4837 (China Unicom) from this list in the top five Autonomous Systems we received traffic from. The other Autonomous Systems we received most traffic from are AS49505 (Selectel.ru),

AS6697 (Belarus Telecom) and AS51166 (Contabo GmbH). These Autonomous Systems do correlate to the top countries in terms of network flows (Figure A.4).

The top fifteen Autonomous systems are combined responsible for 53.77% of the total number of network flows as shown by Figure A.7.

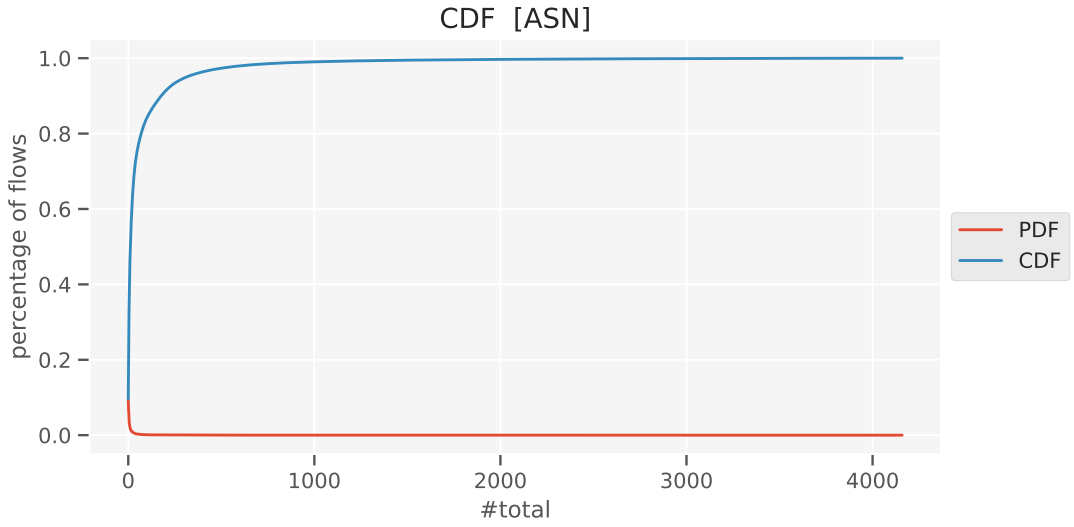

Figure A.7: Cumulative distribution function of the AS targeting the honeypot

A.3 Top IP addresses

A small number of IP addresses is responsible for a large part of the total traffic. In our estimation, we discovered that only 40 IP addresses were responsible for 29.54% of the total network flows.

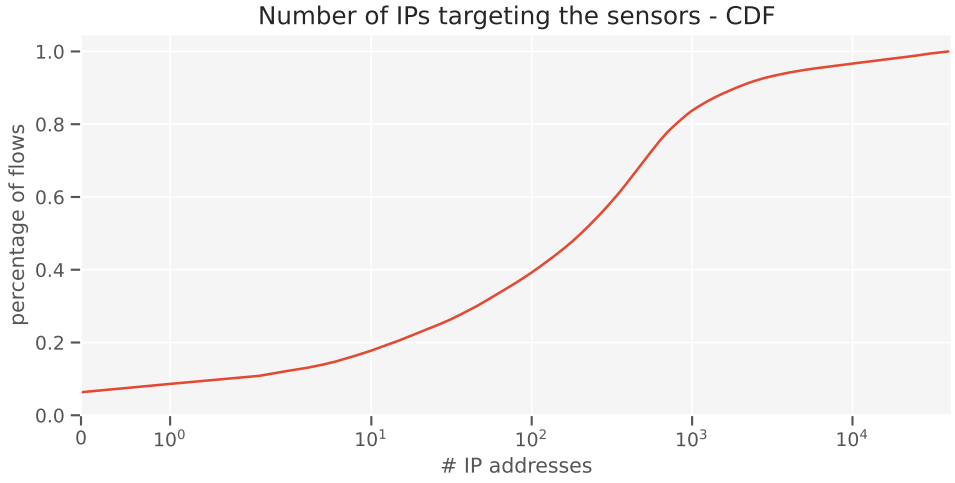

Figure A.8: Cumulative distribution function of the IPs targeting the honeypot

Appendix B

Evaluating Custom Debugging Rules

B.1 Updated Proof-of-Concept

```
1  # Only after resetting the router or using the router for
       the first time, can the script work effectively!
2  import requests
3
4  headers = {
5      "Host": "192.168.0.1",
6      "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:78.0)
           Gecko/20100101 Firefox/78.0",
7      "Accept": "*/*",
8      "Accept-Language": "en-US,en;q=0.5",
9      "Accept-Encoding": "gzip, deflate",
10     "Content-Type": "text/plain",
11     "Content-Length": "78",
12     "Origin": "http://192.168.0.1",
13     "Connection": "close",
14     "Referer": "http://192.168.0.1/",
15     # This assumes the username/password to be admin admin
16     "Cookie": "Authorization=Basic YWRtaW46YWRtaW4="
17 }
18
19 payload = "a" * 512 + "b" * 1024
20 formdata = "[/cgi/auth#0,0,0,0,0,0#0,0,0,0,0,0]0,3\r\nname
       ={}\r\noldPwd=admin\r\npwd=lys123\r\n".format(payload)
21
22 url = "http://192.168.0.1/cgi?8"
23
24 response = requests.post(url, data=formdata, headers=headers
       )
25 print(response.text)
```

B.2 Crash log

```
1  [  630.805275] firmadyne: inet_accept[PID: 2124 (httpd)]:
2  [  630.805531] firmadyne: sys_setsockopt[PID: 2124 (httpd)]:
       fd:18, level:1, optname:20
3  [  630.805797] firmadyne: sys_setsockopt[PID: 2124 (httpd)]:
       fd:18, level:1, optname:21
4  [  630.806062] firmadyne: sys_setsockopt[PID: 2124 (httpd)]:
       fd:18, level:6, optname:1
5  [  630.811293] firmadyne: inet_accept[PID: 2124 (httpd)]:
6  [  630.812708] firmadyne: sys_socket[PID: 2124 (httpd)]:
       family:2, type:1, protocol:0
7  [ dm_checkString ] 1301:  String parameter AdminName's value
       (
8       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
9       aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
10      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
11      aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
12      aaaaaaaaaaaaaaaaaaaaaaaaa-
13 [ dm_setParamNodeString ] 1528:  Parameter's value is not
       valid.
14
15 [ dm_fillObjByStr ] 1976:  Set parameter AdminName's value
       to object error.
16 [  630.816877] potentially unexpected fatal signal 11.
17 [  630.817078] CPU: 0 PID: 2124 Comm: httpd Tainted: G
              W       4.1.17+ #18
18 [  630.817299] Hardware name: Generic DT based system
19 [  630.817460] task: ee1b6c00 ti: ee142000 task.ti: ee142000
20 [  630.817638] PC is at 0x62626262
21 [  630.817745] LR is at 0xb6e49824
22 [  630.817860] pc : [<62626262>]    lr : [<b6e49824>]    psr
       : 60000010
23 [  630.817860] sp : beefe648  ip : b6dd7edc  fp : 62626262
24 [  630.818231] r10: 62626262  r9 : 62626262  r8 : 62626262
25 [  630.818401] r7 : 62626262  r6 : 62626262  r5 : 62626262
       r4 : 62626262
26 [  630.818605] r3 : 000007b8  r2 : beefdf6c  r1 : b6ed696e
       r0 : 0000232f
27 [  630.818813] Flags: nZCv  IRQs on  FIQs on  Mode USER_32
       ISA ARM  Segment user
28 [  630.819060] Control: 10c5387d  Table: 6e26c059  DAC:
       00000015
29 [  630.819392] CPU: 0 PID: 2124 Comm: httpd Tainted: G
              W       4.1.17+ #18
30 [  630.819610] Hardware name: Generic DT based system
```

```
31 [  630.819811] [<c001c8dc>] (unwind_backtrace) from [<
   c0019c70>] (show_stack+0x10/0x14)
32 [  630.820066] [<c0019c70>] (show_stack) from [<c002e6d4>] (
   get_signal+0x41c/0x47c)
33 [  630.820305] [<c002e6d4>] (get_signal) from [<c00194a8>] (
   do_signal+0x8c/0x35c)
34 [  630.820538] [<c00194a8>] (do_signal) from [<c00198d8>] (
   do_work_pending+0x54/0xac)
35 [  630.820778] [<c00198d8>] (do_work_pending) from [<
   c0016c8c>] (work_pending+0xc/0x20)
36 statSetsockopt
37 : Protocol not available
38 [ stat_getsockopt ] 219:  257 opt error
```

Appendix C

Evaluating Karonte

C.1 Testing reachability - example code

```
1  #include <stdio.h>
2  #include <string.h>
3  int main( ) {
4
5      char str[100];
6
7      printf( "Enter a value :");
8      scanf("%s", str);
9
10     printf( "\nYou entered: %s", str);
11
12     // add a memcmp-like function for Karonte to identify
13     int a = memcmp(str, "http", 4);
14     if (a == 0) {
15         printf("\nThis is equal to http :-(");
16     }
17
18     char cpystr[15];
19     // potential buffer overflow :-)
20     if (a != 0) {
21         strcpy(cpystr, str);
22         printf("\ncopied_str: %s", cpystr);
23         printf("\noriginal_str: %s", str);
24     }
25     return 0;
26 }
```

C.2 Testing reachability - full path constraint

```
1  [<Bool (taint_buf_rdi__0_64[7:0] .. taint_buf_rdi__0_64
       [15:8] .. taint_buf_rdi__0_64[23:16] ..
       taint_buf_rdi__0_64[31:24]) == 0x68747470
2  || (taint_buf_rdi__0_64[7:0] .. taint_buf_rdi__0_64[15:8] ..
        taint_buf_rdi__0_64[23:16] .. taint_buf_rdi__0_64
       [31:24]) < 0x68747470
3  || (taint_buf_rdi__0_64[7:0] .. taint_buf_rdi__0_64[15:8] ..
        taint_buf_rdi__0_64[23:16] .. taint_buf_rdi__0_64
       [31:24]) > 0x68747470>,
4
5  <Bool ((if taint_buf_rdi__0_64[31:24] == 112 &&
       taint_buf_rdi__0_64[23:16] == 116 && taint_buf_rdi__0_64
       [15:8] == 116 && taint_buf_rdi__0_64[7:0] == 104
6  then
7      0
8  else
9      (if 0x68747470 <= (taint_buf_rdi__0_64[7:0] ..
          taint_buf_rdi__0_64[15:8] .. taint_buf_rdi__0_64
          [23:16] .. taint_buf_rdi__0_64[31:24])
10     then 0
11     else 255)) ..
12     (if taint_buf_rdi__0_64[31:24] == 112 &&
          taint_buf_rdi__0_64[23:16] == 116 &&
          taint_buf_rdi__0_64[15:8] == 116 &&
          taint_buf_rdi__0_64[7:0] == 104
13     then 0
14     else
15         (if 0x68747470 <= (taint_buf_rdi__0_64[7:0] ..
             taint_buf_rdi__0_64[15:8] .. taint_buf_rdi__0_64
             [23:16] .. taint_buf_rdi__0_64[31:24])
16         then 0
17         else 255)) ..
18         (if taint_buf_rdi__0_64[31:24] == 112 &&
             taint_buf_rdi__0_64[23:16] == 116 &&
             taint_buf_rdi__0_64[15:8] == 116 &&
             taint_buf_rdi__0_64[7:0] == 104
19         then 0
20         else
21             (if 0x68747470 <= (taint_buf_rdi__0_64[7:0] ..
                taint_buf_rdi__0_64[15:8] ..
                taint_buf_rdi__0_64[23:16] ..
                taint_buf_rdi__0_64[31:24])
22             then 0
23             else 255)) ..
```

```
24          (if taint_buf_rdi__0_64[31:24] == 112 &&
               taint_buf_rdi__0_64[23:16] == 116 &&
               taint_buf_rdi__0_64[15:8] == 116 &&
               taint_buf_rdi__0_64[7:0] == 104
25          then 0
26          else
27              (if 0x68747470 <= (taint_buf_rdi__0_64[7:0] ..
                   taint_buf_rdi__0_64[15:8] ..
                   taint_buf_rdi__0_64[23:16] ..
                   taint_buf_rdi__0_64[31:24])
28              then
29                  (if taint_buf_rdi__0_64[7:7] == 0 && (
                       taint_buf_rdi__0_64[6:0] ..
                       taint_buf_rdi__0_64[15:8] ..
                       taint_buf_rdi__0_64[23:16] ..
                       taint_buf_rdi__0_64[31:24]) <= 0x68747470
30                  then 2
31                  else 1)
32              else 255)))
33  != 0x0>]
```