University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Cooperative MTD Framework for IoT Devices

Steven Näf
Zurich, Switzerland
Student ID: 15-710-254

Supervisor: Jan von der Assen, Dr. Alberto Huertas Celdrán
Date of Submission: January 29, 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

The Internet of Things (IoT) offers many advantages to our society, including benefits regarding the economy and human convenience. While these are not empty promises, IoT devices have the major drawback of being inherently vulnerable to malware due to various characteristics. As the number of IoT devices is expected to triple by 2030, possible defense mechanisms against such malware (e.g. Bashlite or Mirai) are essential. This thesis proposed and implemented a prototype of a cooperative and reactive Moving Target Defense (MTD) framework that exploits the weaknesses of Bashlite, a well-known IoT malware. The first weakness is the ability to disrupt the connection of a Bashlite client from the Bashlite server by changing the client's IP address. The second vulnerability is that Bashlite scans and distributes itself via the Telnet port 23. Hence, the infected device is instructed to change its local IP address to disconnect itself from the Bashlite server, and the other devices in the network are instructed to temporarily move their Telnet service port to hide until Bashlite is rendered harmless.

Three different evaluation scenarios were created, all consisting of two virtual machines, one of which is infected with Bashlite that attempts to infect the second machine. The scenarios differed in the inclusion of the cooperative component and the trigger of the execution of the MTD techniques. The two possibilities for the trigger were proactive (every minute) and reactive (after the detection of Bashlite). The evaluation scenarios have shown that the proposed cooperative and reactive framework and techniques have significant advantages over a non-cooperative and reactive approach and a cooperative but proactive approach. In addition to halving the overall infection time in the system, the overall availability of the machines, defined by outgoing packet losses and outgoing and incoming Telnet connections, was also significantly improved. In addition, the CPU and RAM usage of the framework and techniques executed were minimal. Although the cooperative and reactive approach provided by far the best results, each MTD approach has its advantages and further research is required to make use of this promising defense mechanism.

Das Internet der Dinge (Internet of Things, IoT) bringt unserer Gesellschaft viele Vorteile, unter anderem im Bereich der Wirtschaft und des menschlichen Komforts. Das sind zwar keine leeren Versprechungen, allerdings haben IoT Geräte den grossen Nachteil, dass sie aufgrund verschiedener Eigenschaften stark anfällig für Schadsoftware sind. Da sich zusätzlich die Zahl der IoT-Geräte bis 2030 voraussichtlich verdreifachen wird, sind mögliche Abwehrmechanismen gegen solche Schadsoftware (z. B. Bashlite oder Mirai) von entscheidender Bedeutung. In dieser Thesis wurde ein Prototyp eines Frameworks für ein kooperatives und reaktives Verteidigungssystem (Moving Target Defense, MTD) vorgeschlagen und implementiert. Dieser Verteidigungsmechanismus nutzt die Schwachstellen von Bashlite, einer bekannten IoT Malware, aus. Die erste Schwachstelle ist die Möglichkeit, die Verbindung eines Bashlite-Clients mit dem Bashlite-Server zu unterbrechen, indem die IP-Adresse des Clients geändert wird. Die zweite Schwachstelle besteht darin, dass Bashlite den Telnet-Port 23 von anderen Maschinen scannt und sich darüber auch verbreitet. Wegen diesen zwei Schwachstellen wird die infizierte Maschine angewiesen, die lokale IP-Adresse zu ändern, um sich vom Bashlite-Server zu trennen, und die anderen Geräte im Netzwerk werden angewiesen, ihren Telnet-Service-Port vorübergehend auf einen anderen Port zu legen, bis Bashlite unschädlich gemacht ist.

Drei unterschiedliche Bewertungsszenarien wurden erstellt, wobei alle aus zwei virtuellen Maschinen bestehen, von denen eine mit Bashlite infiziert ist, die dann versucht die zweite Maschine zu infizieren. Die Szenarien unterschieden sich durch die Einbeziehung der kooperativen Komponente und durch den Auslöser für die Ausführung der MTD-Techniken. Die beiden Möglichkeiten für den Auslöser waren proaktiv (jede Minute) und reaktiv (nach der Erkennung von Bashlite). Diese Bewertungsszenarien haben gezeigt, dass das vorgeschlagene Framework und die Techniken (kooperativ und reaktiv) erhebliche Vorteile gegenüber einem unkooperativen und reaktiven Ansatz und einem kooperativen, aber proaktiven Ansatz haben. Neben der Halbierung der Gesamtinfektionszeit im System wurde auch die Verfügbarkeit der Maschinen, definiert durch ausgehende Paketverluste und ausgehende und eingehende Telnet-Verbindungen, insgesamt deutlich verbessert. Darüber hinaus war die CPU- und RAM-Auslastung des Frameworks und der ausgeführten Techniken minimal. Obwohl der kooperative und reaktive Ansatz bei weitem die besten Ergebnisse lieferte, hat jeder MTD-Ansatz seine Vorteile, und weitere Forschung ist erforderlich, um diesen vielversprechenden Abwehrmechanismus entsprechend zu nutzen.

Acknowledgments

I would like to thank my two supervisors, Jan von der Assen and Dr. Alberto Huertas Celdrán. They have helped me with all kinds of problems, given much appreciated input when it was not clear how to proceed, and provided great feedback overall. All of this has contributed greatly to the quality of this thesis and also helped me going through the more difficult times while writing the thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

In today's world, IoT devices are already ubiquitous and can be found in several major end-use industries such as healthcare, manufacturing or banking [1]. In 2021, there were 11.3 billion IoT devices worldwide and this number is expected to triple by 2030 [2]. These devices facilitate the integration of the physical world into a computer-based system, providing various benefits such as improved efficiency and economic advantages [3]. However, alongside the many benefits, there are also major security concerns, as the devices are popular targets for malware for a number of reasons. These include poor maintenance by manufacturers and weak/repetitive device passwords [4], [5]. Infected IoT devices are ideal as bots for botnets, which can then launch e.g. distributed denial of service attacks. [6] detected 105 million attacks on its honeypots in the first half of 2019, which is a significant increase from the 12 million attacks registered in the first half of 2018. The security problems, the rapidly growing number of devices worldwide and the increasing number of attacks make a strong defence option indispensable.

One possible defence solution is Moving Target Defence (MTD). This is a cybersecurity paradigm that was first proposed in 2009 [7]. Its aim is to constantly change the attack surface of a target to diminish the probability of a successful attack [7]. Examples include dynamic network techniques, which change network properties such as a device's IP address, or dynamic data techniques, which aim to change data representations [8].

IoT malware such as Mirai or Bashlite include a spreading functionality [9]. This allows the malware to quickly spread to new devices, which in turn increases the power of the botnet at hand. This thesis aims to take this spreading functionality into account and create a cooperative MTD framework that mitigates/prevents the malware in a more effective way than a non-cooperative MTD framework is capable of.

1.2 Description of Work

This thesis proposes, implements and evaluates a cooperative MTD framework with two different MTD techniques. The framework is based on [10], but includes a cooperative component to mitigate the selected malware and prevent its spreading. The selected malware is Bashlite, a well-known command and control malware, which was modified and completed for this thesis. After analyzing the weaknesses of Bashlite for possible levers where MTD techniques can be applied, two MTD techniques are selected and implemented. The first is to change the IP address of the infected device, which permanently disrupts communication with the Bashlite control server. The second technique is to temporarily change the Telnet service port of other susceptible machines in the network. As many IoT malware infect other devices via the Telnet port, temporarily moving the Telnet service port of the susceptible devices prevents the devices from being found in the first place. After a specified number of seconds, when the IP address change has finished securing the infected device, the Telnet service port changes back to port 23.

The framework and techniques are evaluated in three different scenarios using three different metrics. These metrics include the overall infection time of the system, the interruption of the availability of the machines and the CPU and RAM usage needed by the MTD framework and its techniques. The base case for the evaluation is a non-cooperative and reactive MTD framework that only uses the IP address change to clear the devices after Bashlite is found on them. Reactive means that there is a detection mechanism for the malware and the MTD techniques start as soon as the malware is found on the system. So in the first scenario, each machine has to be infected before it can do anything against Bashlite. The second scenario is cooperative and reactive. As soon as Bashlite is found on a machine on the network, the infected machine initiates the IP address change and all other machines in a given IP range change their Telnet service port to a different port. This is the solution presented in this thesis. The third scenario is cooperative, but proactive. Proactive means that the MTD techniques are run at a specified interval, in this case 60 seconds. Following the evaluation, its results are presented and discussed in-depth.

1.3 Thesis Outline

After this introductory chapter, the thesis continues with the necessary background information (Chapter 2) for the rest of this thesis. Various information is presented there, including a general overview of MTD and why IoT devices are susceptible to malware. This chapter is followed by the related research chapter (Chapter 3), which first presents the state of research of MTD in IoT, then introduces some MTD IoT frameworks, and finally introduces some MTD IoT mechanisms/techniques. Chapter 4 is the implementation chapter, which deals with the selection and adaptation of Bashlite and possible levers where MTD techniques could be applied to mitigate a Bashlite infection. Additionally, the final implementation is presented in detail in the implementation chapter. This final implementation is then evaluated in Chapter 5, which first introduces the methodology of the evaluation and then presents the results. The last two chapters of the thesis are

the discussion and the conclusion. Additionally, some limitations are presented in the discussion and possible future research in the conclusion.

Chapter 2

Background

This chapter provides an overview of the background knowledge relevant for the thesis. First, a general overview of Moving Target Defence (MTD) is introduced. Second, some background information on IoT security is presented. This includes some architectural information as well as common IoT malware and how they work. A subsection on cooperative defence concludes the chapter.

2.1 General Overview of Moving Target Defense

Although the idea of modifying system components to prevent others from disrupting the systems has been around for some time, the first use of the term "MTD" was proposed in 2009 by Networking and Information Technology Research and Development (NITRD) [7]. The following paragraph is based on this very NITRD source [11], which despite its age serves as a good overview of MTD.

One of the main problems of systems is their relatively static configuration. This static design stems from the fact that, in the past, system requirements focused on simplicity and elegance rather than security, as exploitation of vulnerabilities was not a concern. An example are IP addresses and other configuration parameters that remain static over a relatively long period of time. An adversary needs to know the vulnerabilities of a system in order to attack it effectively. The longer the vulnerabilities exist, the more likely they are to be exploited. Therefore, a static system is a significant advantage for an adversary because it gives them enough time to prepare and plan their attack. This is the motivation behind MTD, which involves building systems that change rapidly to minimise the likelihood of a successful attack.

Another great and more recent summary was given by [7]. Since system information does not expire, an attacker with enough resources (e.g. time) will ultimately find a vulnerability and succeed in his attack. MTD tries to increase the resources needed for an attacker to succeed by continuously altering the attack surface of the system. The sum of a system's vulnerabilities can be defined as its attack surface [12].

[13] concluded from the existing literature that three elements must be defined for an MTD technique to achieve the defence objective.

1. **What to move**: This defines the moving parameter (MP), which is an essential attribute (e.g. IP address or service port) of an attack target. Each MP has a domain from which its values can be selected.

2. **How to move**: This describes how the MP should be moved. This involves selecting a new MP value from its range and replacing the old MP value. There are several ways to choose the new value, such as randomly, game theoretically or situationally.

3. **When to move**: This describes the frequency with which the current value of the MP should be replaced by the new one. This is a critical element because if the MP changes too often, it could lead to poor system performance; if it changes too little, an attacker could be successful. [7] identified three different decision processes in the literature: time-based, event-based and hybrid.

[8] reviewed five dominant domains of MTD techniques, including the advantages and disadvantages of each. Below is a presentation of these 5 domains. For each domain, [8] also indicated the phase of the attack that the domain seeks to disrupt. For this, the authors chose a five-phase attack consisting of the following phases:

- Reconnaissance: This phase is mainly limited to observation, where the attacker tries to find a target and collect basic information about it. An example of this is finding the IP address of a host through IP scanning.

- Access: In this phase, the adversaries collect detailed information about their target. In the case of e.g. a web server, this could include its operating system or configuration.

- Development: In this phase, the adversaries develop an attack that targets a previously found vulnerability. This can be achieved offline without any connection to the target.

- Launch: In this phase, the adversaries compromise the target by delivering the attack payload. This can be done in a variety of ways, including infected media or over the network.

- Persistence: If the adversaries wish to remain in the compromised system, they can install additional entry points into the system, such as backdoors.

Each of the MTD domains described below attempts to disrupt one or more of these phases. This is only an overview of the domains and does not cover specific techniques [8].

2.1.1 Dynamic Networks

MTD techniques in this domain modify network properties to increase the required workload for an attacker and thus reduce the probability of a successful attack. These techniques are primarily used to prevent the success of the reconnaissance phase, but can also be used to prevent a successful launch phase. Possible means to achieve these network modifications include frequent address and port changes, or changing the logical network topology. The logical network topology is how devices are arranged in a computer network and how they communicate with each other [14].

One problem with such techniques is that they may hinder convenient use of the system when applied to a service of a system that needs to remain in a known network location. An example are public servers, where such a technique would defeat its purpose as the server would be hidden even for legitimate users. Another challenge is that the degree of uncertainty created for attackers is critical to the effectiveness of randomization. Entropy, a value that depends on the number of possible values and their corresponding probabilities, can measure the uncertainty in a random value, but this entropy is limited in many dynamic network techniques. One reason for this is that IP addresses and port numbers can be limited by the network infrastructure.

2.1.2 Dynamic Platforms

MTD techniques in this domain focus on modifying the computing platform characteristics. The goal is the disruption of attacks that depend on specific platform properties. There are several properties that can be changed, such as the operating system, storage systems, virtual machine instances, or communication channels. It is possible for the same application to run in parallel in different architectural contexts, or for applications to migrate from machine to machine. In terms of attack phases, these techniques provide protection in all five phases, but are most beneficial in the access, development and persistence phases. This is because an attack is much more difficult if it requires exploits for multiple platforms to succeed. This is especially true if the program is executed in parallel on several instances.

A problem with dynamic platforms is that these techniques also increase the attack surface because additional code is used to control and manage migrations. Additionally, an attacker may gain an advantage due to a specific vulnerability in the platform on which the application is currently running. Another problem with dynamic platforms is that it can be difficult to maintain or synchronise state across platforms in a platform-independent format if required by the application. Even if the state transfer can be performed, care must be taken to ensure that the attacker does not remain in the state, resulting in persistence of the attacker in the system.

2.1.3 Dynamic Runtime Environment

The goal of dynamic runtime environment techniques is to prevent the exploitation of software vulnerabilities by randomizing the environment in which the application runs.

These techniques assume that the attacker already has an exploitable vector, and the techniques are designed to prevent the attack from being executed. One problem that these techniques can prevent is buffer overflow exploits, which can lead to code injection. These techniques can be further divided into two sub-domains: Instruction Set Randomization (ISR) and Address Space Randomization (ASR). ISR can take place in the application, the operating system, or the hardware, and its goal is to prevent attackers from predicting how the program will execute. This is done by randomizing the instructions in an application. A specific example of ISR is encrypting instructions at load time and decrypting them just before execution. ASR techniques aim to create a randomized memory layout from a deterministic one. This prevents the possibility of using a known memory address for things like control flow redirection. One of the best known and most widely used runtime techniques is address space layout randomisation (ASLR), which falls into the ASR subdomain.

Both subdomains have weaknesses, in the case of the ASR subdomain, or more specifically ASLR techniques, it is the fact that typically only a part of the memory of the application is randomized, while the other part remains static. This static part may be sufficient for an attacker to develop a meaningful payload. Another related problem is that only the base address of the memory segment is randomized, but the relative addresses remain unchanged. This means that the attacker can bypass ASLR by using relative addresses. In addition, due to the architectural limitations of the defended system, the randomized memory spaces are often too small, making them vulnerable to brute-force attacks. Regarding ISR, the main weakness is the performance overhead caused by the fact that ISR techniques often rely on software emulations, as there is often no hardware support. There are ISR techniques that depend on low overhead methods as e.g. XOR encryption, but these techniques are weak encryption methods and there is a possibility that the attacker can recover the key to inject correctly encrypted instructions.

2.1.4 Dynamic Software

The goal of MTD techniques in this domain is to diversify the application without changing its functionality. Equivalent program instructions substitute each other, changing various properties such as the internal data structure layout or the sequence of instructions. This reduces the likelihood of a successful attack, as the attacker must correctly guess the software variant being used. These techniques aim to disrupt the attacks' development and launch phase by creating uncertainty and making code injection and code reuse more difficult. There are several ways to apply these techniques, either by using an application that has its own internal randomization capability, or by creating multiple semantically equivalent binaries.

In practice, there was no widespread use of dynamic software techniques in 2014. The examples that did exist were mostly limited to academic and research environments. Unfortunately, no more recent source on the current situation of dynamic software techniques could be found. There are also several weaknesses of these techniques, one of which is that it is complex to ensure that the diversified application provides the same functionality as the original. There is also a lack of scalability, the possibility of unexpected side

effects, and the significant performance overhead associated with heavy binary translation and emulation. Finally, many applications are designed for maximum performance, and semantically equivalent applications could reduce this performance.

2.1.5 Dynamic Data

The goal of the dynamic data domain is to change the internal or external data representation by changing the properties of the data representation, such as syntax or format. Similar to the dynamic software domain, the semantic context should remain unchanged, but the change in data representation should prevent unauthorized use or access to the content. These techniques should also complicate the development and deployment phases of the attack, as attack development is hampered by the need to find an adequate payload for the various data representations. Some of these techniques have their origins in techniques developed against data corruption. One example is a technique where computations are performed on multiple data representations, providing the ability to detect corrupted or malicious input. Figure 2.1 shows two different data representations with the same semantics.

These techniques also have weaknesses. One is that most standard binary formats support only one canonical representation, resulting in a lack of diversity in possible data encodings. Another drawback from a practical point of view is that dynamic data techniques increase the effort required for application development as well as runtime performance, since multiple data representations may have to be processed and monitored.

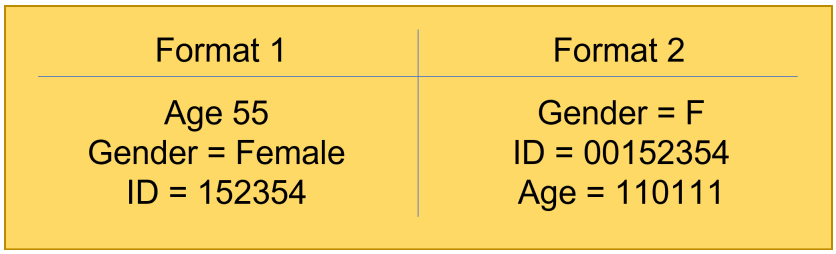


Figure 2.1: Two Different Data Representations With the Same Semantics.

2.1.6 Summary

Table 2.1 shows the different moving target defence domains and the corresponding attack phase they seek to disrupt. In their discussion, [8] identified three critical properties for effective MTD, namely unpredictability, comprehensiveness and timeliness. The first property is crucial because if the attacker can predict the next movement of a defence mechanism, the defence becomes obsolete. Comprehensiveness means the inclusion of all elements that could work against an attack, as one element alone may not be very useful. An example is the case where the location of an application's library is randomized, but the application remains in a fixed location. This means that attackers can simply ignore the randomization and attack the fixed code. Timeliness is also important. For

example, if attackers can observe the outcome of a moving target defence technique, this knowledge could give them an opportunity to attack. It is also crucial that the attacker is exposed to the diversity of the environment within the attack time. The authors gave an example where an application is migrated among three platforms, so that the attackers would need another vulnerability to continue the attack after the migration to the new platform. However, if the attack time is smaller than the migration time, the security is reduced because the attackers have three different platforms from which to choose vulnerabilities. The authors conclude that some techniques work better for general-purpose environments and some better for specific purposes, as each technique has different strengths and weaknesses.

MT Domains	Attack Phases				
	Reconnaissance	Access	Development	Launch	Persistence
Dynamic Networks	x			x	
Dynamic Platforms		x	x		x
Dynamic Runtime Environments			x	x	
Dynamic Software			x	x	
Dynamic Data			x	x	

Table 2.1: A Summary of the Moving Target Defense Domains and the Corresponding Attack Phase They try to Hinder [8].

2.2 Internet of Things

[3] gave a brief introduction about what the Internet of Things (IoT) is. It is about the collection and exchange of data through networked items embedded with electronics, sensors, software, actuators and a network connection. Examples of these items are physical devices, vehicles or buildings. These items can be controlled or sensed across an existing network infrastructure. This enables better integration of the physical world into computer-based systems, which has several benefits, such as reduced need for human intervention and economic benefits. But there are also major challenges for IoT, two of which are security and privacy. For example, a smart meter in a house knows when someone is at home, and this data is also shared with other devices and databases by companies that therefore also have this information.

2.2.1 Current and Future Distribution of IoT Devices

According to [2], there existed 11.3 billion IoT devices worldwide in 2021. By 2030, this number is expected to nearly triple to an estimated 29.3 billion devices. [1] estimated

the global market size for IoT from 2022 to 2029. They estimated the market size to be 2,465.26 billion in 2029, which is significantly higher than the current 478.36 billion. Also interesting is the analysis of the current market share by end-use industry. At 21.5%, the largest share of the end-use industry is accounted for by the healthcare market, followed by the manufacturing market and the IT & telecom market. These three account for around 50% of the total market share. Other examples include retail, transportation and banking, financial services and insurance.

[15] has published a survey conducted in July 2021 on Smart Home solutions (consisting of IoT devices) and how many users are currently using them and how many users intend to use them in the future. 5 different groups were formed, which are shown in Table 2.2. It can be seen that some groups currently have a higher share and others a lower share, but the intended future use is significantly higher for each group. Another interesting insight is the breadth of the groups, ranging from lamps to security to garden and balcony. Thus, IoT devices can already be found in various places, and they will become even more important than they are today.

		Current		Future	
Category	**Subcategory**	**Germany**	**UK**	**Germany**	**UK**
Lamps and Lighting		13%	11%	43%	47%
Heating and Air Conditioning		8%	10%	41%	49%
Security Solutions	Smart Cameras	7%	13%	37%	49%
	Smart Sensors	12%	11%	48%	56%
Appliances	Small Home Appliances	4%	3%	28%	24%
	Large Home Appliances	4%	3%	32%	35%
	Smart Vacuum Cleaners	8%	2%	30%	23%
Garden and Balcony	Smart Robotic Lawnmower	3%	1%	21%	18%
	Smart Irrigation Systems	3%	1%	27%	22%
Overall		**16%**	**25%**	**41%**	**55%**

Table 2.2: Results of a Survey Regarding IoT Devices in Households Based on Data in Germany and the UK.

2.2.2 Architecture

[16] published an article on a general overview between architectures, protocols and applications of IoT. The authors presented three different architectures that are most commonly found in the literature.

The first architecture is a generic high-level architecture consisting of three layers, namely perception, network and application. The perception layer is the physical layer that inter-

acts with the environment through information gathering and processing. Objects with computing power and the ability to interact with the outside world are an integral part of this layer. The network layer is the communication layer responsible for transferring data from the perception layer to the application layer. This layer includes all protocols and technologies required for this connection. An example of a protocol is 6LoWPAN, which stands for IPv6 over Low power Wireless Personal Area Network. The personal area network connects devices in a user's immediate environment, such as Bluetooth headphones with a mobile phone [17]. The final layer is the application layer, which contains the essential software for a specific service. The data from the preceding layers is stored, processed, aggregated and filtered in this layer. The processed data is then made available to the IoT application.

The second architecture is the service-oriented architecture. This architecture extends the three-tier architecture by adding a service layer between the application layer and the network layer. This service layer provides services to support the application layer and consists of several sub-services. The goal of this service-oriented architecture is to enable software and hardware reuse and to coordinate services. The architecture helps to connect different functional units through protocols and interfaces.

The third common architecture is the middleware architecture, also known as the five-layer architecture. This architecture is made up of five layers: the perception layer, the network layer, the middleware layer, the application layer and the business layer. The middleware layer, which aggregates and filters data received from the hardware, is an important layer. Various technologies are hidden in the middleware and standard interfaces are provided. This means that developers do not have to worry about compatibility between the infrastructures and the application and can therefore focus on the application development.

The three architectures discussed can be seen in Figure 2.2. Some additional architectures that also exist are the cloud-based architectures or the edge computing-based architectures.

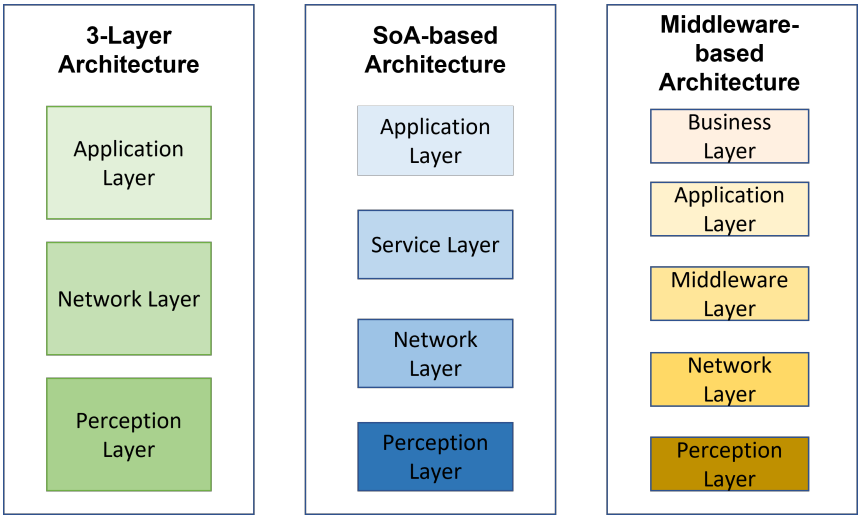


Figure 2.2: The Three Most Common IoT Architectures [16].

2.2.3 Hardware and Software Details of IoT

[18] published a report on an IoT & Edge Developer Survey with interesting insights. This survey was conducted in 2022 and 910 developers, committers, architects and decision makers were interviewed. One of the key findings for this thesis was that security concerns have almost doubled this year and are now in the top three challenges for developers, along with data collection & analysis and connectivity. The most commonly used programming languages for constrained devices are Java, C and C++. In terms of the operating system (OS) for constrained devices, Linux distributions are the top choice with 43%. FreeRTOS is in second place with 22%. FreeRTOS is a real-time operating system for microcontrollers that is freely distributed under the MIT Open Source Licence [19]. In third place is No OS/bare metal for constrained devices. The Linux operating system is further broken down into its distributions. Ubuntu makes up 23% of all Linux operating systems, the second distribution is Raspbian with 20%, the third is Alpine with 18% and the fourth is Debian with 17%. Interestingly, there are many more distributions in use, but they have a maximum share of 13%. This information can be seen graphically in 2.3.

In terms of architecture for constrained devices, ARM dominates. The most common architecture is ARM Cortex-M0/m0+ with 26%. ARM Cortex-M3/ARM Cortex-M4 follows with 24%. In third place is the ARM Cortex-M7 with 20%.

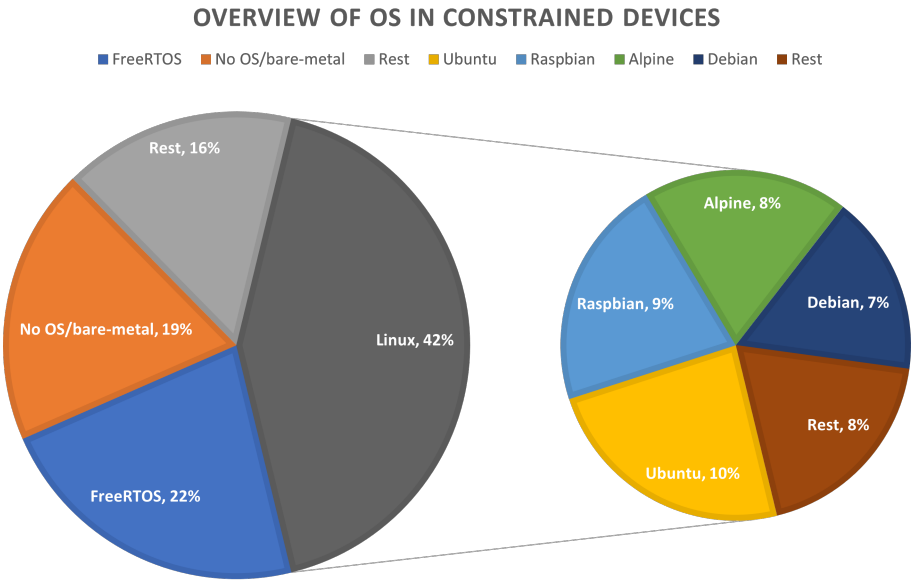


Figure 2.3: The Most Commonly Used OSs in Constrained Devices.

2.3 Malware and IoT

This section gives a general overview of malware, especially in the context of IoT devices. A more detailed analysis can be found in Section 4.3, where Bashlite is analysed.

Malware or malicious software is a term used to describe malicious code or a program that is harmful to a system [20]. Examples of malware or malware families related to IoT are Mirai, Bashlite, Tsunami or Hajime [21]. Mirai and Bashlite are probably two of the best known examples. Mirai, for example, was still highly prevalent in the first quarter of 2019 [6]. Bashlite, also known as Gafgyt [22], was already much less common in 2019 [6], but other malware such as Mirai has inherited from its source code [21].

[9] gave a brief overview of Bashlite and Mirai and how they are used to create botnets. The rest of this paragraph is based on that article. Since the source code of Mirai is based on that of Bashlite, they share some similarities. For example, both infect IoT devices that can be accessed with known and/or vulnerable authentication credentials. They also both aim to create botnets. These botnets have different components. The command and control (C&C) servers send commands to the infected devices and essentially act as the operator's interface to the botnet. The bots are the infected devices that make up the botnet. They execute the commands received from the C&C servers. There are also scanners that identify vulnerable devices by looking for Telnet and SSH servers that the scanners attempt to log into. Loaders download and run the malware of the botnet after logging into the vulnerable devices. Malware servers provide resources such as executable binaries to the botnet. A possibly distributed database stores collected information such as scan results or active bots.

Such botnets can be used for different purposes [5], which are described below. The first purpose is the aforementioned DDoS attack. The idea is to attack a target by sending so much traffic from many different machines that the target cannot handle the volume, eventually taking the target down. Another use for botnets are spam bots. Using botnets, spam can be sent from IP addresses that are not yet known to be spam relays and therefore not yet blocked by system administrators. A third purpose are crypto mining bots, which can be used to mine cryptocurrencies such as Monero. These malware require a device with sufficient processing power (e.g. smartphones), so constrained devices are not optimal for such malware to infect.

According to [4], there are five main reasons why IoT devices are advantageous for creating botnets:

- IoT devices often operate around-the-clock, they do not have on-off cycles like laptop and desktop computers.
- IoT devices are poorly maintained. A very common problem is that devices are set up and then forgotten about as long as they are working properly.
- IoT devices are capable of generating significant distributed denial of service (DDoS) attack traffic, similar to the attack traffic generated by modern desktop systems.
- IoT devices are often either non-interactive or require minimal user intervention, resulting in infections going unnoticed.
- IoT vendors favour usability and user-friendliness over security, resulting in weak protection.

The last item in the enumeration includes a number of known vulnerabilities [5]. The first are weak passwords to make it easier to set up and use the device. These passwords are many times printed in the user manual or even outside the packaging. As the credentials are often the same for all the same devices, this is a huge problem. Even if the credentials were not so accessible, they are often predictable combinations such as "admin/admin". In addition, many devices lack encryption. Such security features are often not even considered. In addition, vendors sometimes add hidden access mechanisms, called backdoors, to devices. They might do this to make it easier to support the device, but it could also be used by hackers. An example is an open port on the device that cannot be closed.

2.3.1 IoT Malware Statistics

[6] published a report in which the authors researched attacks on IoT devices using honeypots. The authors mentioned that such honeypots are the best option to track attacks, catch malware or just get a general overview. This subsection is based on that source. There exist three different common types of honeypots.

- Low-interaction honeypots simulate services such as SSH, Telnet and web servers. The attacker is fooled into thinking that this is a real susceptible system and attacks the honeypot.
- High-interaction honeypots are real systems that have the advantage of running fully POSIX capable systems. POSIX is a standard defined by the IEEE [23]. This standard helps to maintain compatibility between operating systems [24]. As these honeypots are real systems, it is important to take action against the malicious activity of the malware (e.g. prevent further systems from being compromised).
- Medium-interaction honeypots are a combination of low-interaction and high-interaction honeypots.

The authors collected data from more than 50 honeypots around the world over the course of more than a year.

In the first six months of 2019, the Telnet honeypot detected more than 105 million attacks from 276,000 attacking IP addresses. This is significantly more than the 12 million attacks from 69,000 IP addresses in the first half of the previous year. The most attacking IP addresses came from Brazil and China with 30% and 19% respectively. Egypt, Russia and the US followed with 12%, 11% and 8% respectively. Regarding the top malware threats that attacked the Telnet honeypot, 6 out of 10 were Mirai variants. There are several reasons for this, including the public availability of the malware and its ability to create bots of any complexity and for any hardware configuration. The NyaDrop family of malware was the most common, accounting for around 39% of all attacks. This is a Linux trojan that targets IoT devices and specifically the MIPS CPU architectures [25]. In second, third, fourth and fifth place were the Mirai variants with 22%, 12%, 2% and 2% respectively.

Another interesting finding was the most common combination of credentials tried. "support/support" was tried 2,627,805 times, "root/vizxv" was tried 2,376,654 times, and "admin/admin" was tried 2,359,985 times in the first quarter of 2019. The second is the default combination for connecting to a vulnerable camera from Dahua via Telnet [26]. The last interesting finding the authors made was in terms of the ports targeted by the malware. TCP ports were clearly the most targeted, with only a small number of attacks targeting ICMP and UDP ports. Unfortunately, the authors did not provide statistics for other protocols such as SSH in the same depth as they did for the Telnet protocol.

2.3.2 P2P IoT Botnets

The botnets described so far correspond to the typical IoT botnet [27]. Adversaries control the botnet from a C&C server that controls various infected devices. This means that taking out the C&C servers renders the entire botnet useless, regardless of how many bots are connected to the system [27]. While taking down many C&C servers can be cumbersome, it can also be a convenient solution against botnets. For example, this strategy was used against the Andromeda botnet [28]. Several international authorities took action against domains and servers that were spreading the Andromeda malware. This involved, for example, sinkholing 1500 domains. Sinkholing is when traffic is redirected to a server other than the intended one, such as one controlled by law enforcement authorities [28]. However, this solution vanishes as soon as the botnet is set up as a peer-to-peer (P2P) network. [29] wrote a technical briefing on the development of IoT botnets in combination with P2P networking. The rest of this subsection is based on that source.

Unlike a botnet with a central server, a P2P network is much more robust. The authors use BitTorrent as an example because it has withstood the test of time, having been used to share illegal content for over 20 years without the authorities being able to shut it down. In the case of a P2P IoT botnet, each bot would need to be disinfected separately, as there is no central server. This is also a problem caused by the insecurities of IoT systems, as discussed in Section 2.3. In the case of a desktop environment, mass cleanup would theoretically still be possible, e.g. by antivirus vendors, but this is not an option for IoT devices because there is no antivirus protection. Fortunately, there are not many P2P botnet families yet, the authors stated that they have only seen five families so far, and they also suspect that they are not common at the moment. Although they do not know the actual number of infections, they find it worrying that the rate at which P2P botnet malware appears is increasing. The authors conclude that this indicates an increased interest in creating P2P botnet malware. The authors then briefly describe each of the five malware families. One of these five is described below to give a general idea.

The most recent malware mentioned by the authors is "HEH". HEH is written in Go and scans and infects via Telnet port 23 and port 2323 with hardcoded credentials and brute-forced passwords. The malware starts by randomly selecting an IP address and then uses an algorithm to derive the P2P port from that IP address. Thus, no list of IP addresses and ports is required. Once a victim is infected, it attempts to connect to other peers. Additionally, the malware stops the HTTP service and starts its own service instead, which acts as a download site for infected devices. The P2P protocol has 5

operation codes, 4 of which are for synchronisation and one for receiving commands. The latter accepts various instructions such as 'exit', 'attack', 'execute' and 'self-destruct'. According to the authors, the last command was particularly interesting because it is unusual for a bot to be able to destroy itself.

The authors also shared their thoughts on the future development of P2P IoT botnets. They emphasized the importance of making money as an incentive for cybercriminals to develop malware. Since these incentives are critical to development, it makes sense to take a look at how these incentives have evolved. Routers are an interesting target for attackers because they act as an entry point into a home network. An infected router also offers many opportunities such as man-in-the-middle attacks or information theft. Additionally, an infected router allows lateral movement to infect other unsecured devices on the network. The authors also mentioned that, in terms of financial incentives, a distinction needs to be made between the pre-Covid 19 era and now, due to the increase in home offices. For example, an infected home router could now be an entry point into a company, which could be a possible financial incentive for adversaries. There is also a general risk that a successful attack will motivate other cybercriminals to do the same. For example, the authors predict that once a P2P IoT botnet is successful enough, all other botnets will start using P2P capabilities as well. As a result, there is a realistic risk that such P2P malware will be further developed and distributed, but this is highly dependent on the financial incentives.

As this chapter has shown, there are many problems with the security of IoT devices. There are several reasons why IoT devices are susceptible to malware, such as poor maintenance or a focus on usability rather than security [4]. Infected IoT devices can be used for malicious behaviour. One problem are botnets, where infected devices act as bots that execute commands sent by an adversary. As [6] noted, there were about 9.5 times more attacks in the first half of 2019 (105 million) than in the first half of 2018 (12 million), which is an extreme increase. Another growing problem are P2P IoT botnets. Even if they are not very prevalent currently, it is certain that these P2P botnets can become a major threat in the future because they are almost unkillable. Each of the above points makes it clear that a way to defend against such threats is essential.

2.4 Cooperative Defense

There were not many resources available on the subject of cooperative defence. The ones found are presented in this section.

[30] is a relatively old paper from 2005 in which the authors aim to detect DDoS attacks in the intermediate network. The goal was to enable a DDoS attack detection system to share information, rather than to improve a currently available detection method. To achieve this, they proposed a dynamic defence infrastructure consisting of independent defence nodes and assumed that a DDoS attack heading towards a victim would consume more bandwidth than a normal use of the Internet. For scalable and resilient communication to exchange attack information, the authors designed a directional gossip mechanism. Each defense node sets a limit on the amount of traffic it deems malicious, based on

its own defense strategy. This usually results in a high number of false positives due to the dynamic nature of the Internet. The defense accuracy is improved by the gossip mechanism, which helps to transmit information between nodes. During this information exchange, the rate limit is adjusted at each individual defence node. When the mechanism for aggregating information converges, the defense node's rate limit will have roughly global information of the attack, which allows it to more accurately block/drop malicious traffic.

[31] and [32] proposed and evaluated a cooperative defence mechanism based on the blockchain. The authors presented a scenario in which a web server in an autonomous system is under a DDoS attack from devices hosted in other autonomous systems. An autonomous system is a network or group of networks which share a single routing policy [33]. A routing policy contains a list of IP addresses controlled by the autonomous system and a list of other autonomous systems to which it is connected [33]. In this scenario, the web server relies on the defences of the autonomous system where the server is located. As it is better to block malicious traffic close to its origin, this approach is not ideal. This is where the blockchain comes in. The idea is to store the attacker's IP address in a smart contract created by the collaborative defence participants. In this way, subscribed autonomous systems on the Ethereum blockchain receive an updated list of addresses to be blocked and also confirm the authenticity of the attack. Once the other autonomous systems receive the updated list and confirm the attack, the mitigation strategies in the autonomous system can be triggered to block malicious traffic close to its origin.

Chapter 3

Related Work

This chapter presents related work. The first section presents an article on the current state of research on MTD in IoT. The second section presents related work in the area of MTD frameworks for IoT devices. The last section gives an overview of specific IoT techniques and explains how they relate to the work at hand.

3.1 State of Research of MTD in IoT

[7] did a literature review analyzing existing MTD for IOT techniques. This section is based on that source. The authors defined four research questions, all of which are of interest in the context of this thesis. The first research question concerned the number of proposals for MTD techniques that exist for IoT. They concluded that since 2013, 32 novel proposals have been proposed for the IoT domain. In contrast, more than 80 different general purpose MTD techniques have been proposed from 2009 to 2018 [34].

The second research question was related to the characteristics that can be observed in MTD techniques for IoT. For this question, the authors created an MTD taxonomy showing the distribution of these 32 techniques grouped in the MTD domain mentioned in Section 2.1. The dominant techniques are the networking techniques with 54%, followed by the dynamic runtime environment techniques with 20%. Software and data techniques follow with 13% and 10%, respectively. Dynamic platform techniques have the smallest share with 3%. Figure 3.1 shows these shares graphically. These shares differ from those of the general-purpose MTD techniques, where, for example, the dynamic network techniques account for only about 21% or the dynamic platform for about 20%. A possible reason for this is that recently there has been an increasing interest in network-based MTD techniques [7].

The third research question was how sound the security foundations of the proposed techniques are. To answer this question, the authors classified each of the 32 techniques into three different cryptographic categories. The first category, which includes twelve techniques, completely lacks cryptography or uses a random process without providing further information about it. The second category either uses cryptographic techniques that are

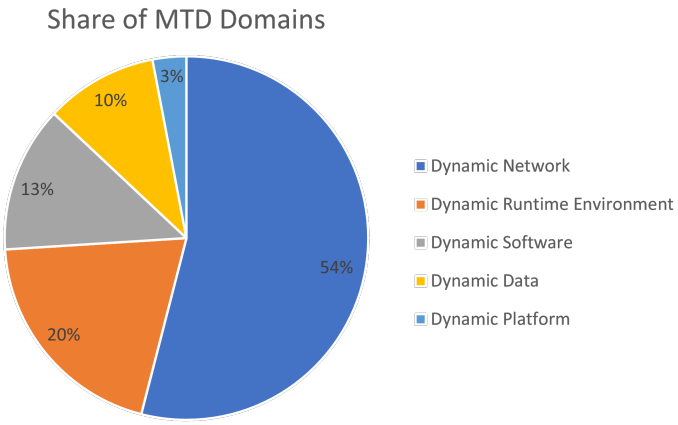


Figure 3.1: Share of IoT MTD Techniques Grouped by MTD Domains [7].

known to be vulnerable or uses custom cryptography without proof. Nine techniques fall into this second category. The last category is the one that has adequate security foundations, such as using state-of-the-art cryptography like SHA256. There are eleven techniques in this category. The authors point out that although they have simplified the measurement of security, 32% is a small value considering that the central goal of MTD is to improve security. The fourth research question is the most interesting and relates to the extent to which the proposals are applicable in a real-world IoT deployment. To answer this, the implementation and evaluation aspects of the proposed techniques were examined by the authors. They concluded that 50% of the proposed techniques show very strong or strong evidence that they can be used in a real IoT deployment. Another 25% of the techniques show mild evidence, and another 25% have weak to no evidence that they can be used in a real IoT deployment. The authors conclude that these are encouraging results.

3.2 MTD IoT Frameworks

[35] presented an MTD framework aimed at mitigating multi-purpose malware. The framework consists of two modules, the MTD decision module, which decides when to deploy an MTD mechanism, and the MTD enforcement module, which decides what MTD mechanism to deploy and how to do that. The authors distinguished between a rule-based proactive approach and a machine-learning based reactive approach to determine when the MTD mechanisms should be executed. They were able to detect normal and malicious behavior in about 10 seconds. [10] wrote a bachelor thesis, whose results are part of [35]. This thesis is about mitigating cyberattacks on resource-constrained devices using MTD. To achieve this, the author developed an architecture consisting of three main components, an *MTD Deployer Server*, an *MTD Deployer Client*, and the MTD solution itself. The *MTD Deployer Client* serves as an interface for external programs to notify the *MTD Deployer Server* of an attack. After the *MTD Deployer Server* receives

an attack report, it searches for the appropriate MTD solution, whereupon the *MTD Deployer Server* launches the appropriate script of the MTD solution to mitigate the attack on the affected machine. This architecture was also the basis for the thesis at hand.

[36] created a framework that helps to answer the fundamental design questions (What, How, When) described in Section 2.1 by using five different variables: attack success probability, system downtime, CPU time, energy consumption, and memory usage. The framework starts with policies, i.e., desirable goals and targets to be achieved. Then some strategy parameters (what to move, when to move, and how to move) are defined. These parameters are then applied to the IoT system which will be continuously attacked. During these attacks, the five variables are measured. These variables are then analyzed using multiple criteria decision analysis. The result of this analysis is compared to the targets defined in the policies. Depending on this comparison, either the final strategy parameters are fixed or different strategy parameters need to be created and the framework starts over. Finally, the authors used this framework to select the most suitable "when to move" parameter for a proposed MTD technique.

[37] proposed a generic MTD framework for IoT called IANVS. This framework consists of four different components that should facilitate the implementation of MTD in distributed systems, since in such systems the parties involved need to agree on the MP value. The four components are

- An authenticated key establishment mechanism called AKE. This provides a secret cryptographic key that only trusted parties of the MTD strategy must know.
- An authenticated state synchronization mechanism called Auth-SYNC. This provides a system state value that must be fresh and authenticated.
- A cryptographically secure pseudo random number generator called CSPRNG. This takes the cryptographic key from the AKE component and the system state from the Auth-SYNC component and generates a "cryptographically secure pseudo random binary key stream".
- A mechanism called MP-Map that outputs values in the MP domains with equiprobability. This takes the keystream from the CSPRNG and maps it to a value in the MP domain (e.g. port hopping).

The authors also applied this framework and designed two MTD techniques for IoT.

3.3 MTD IoT Mechanisms

This subsection briefly introduces MTD mechanisms/techniques. Since the intention is to use dynamic network techniques for the implementation part of this thesis, this subsection is mostly limited to this type of techniques. However, other techniques that were in the same articles as the dynamic network techniques are also briefly mentioned.

[38] proposed "MT6D", which stands for Moving Target IPv6 defense, with the goal of protecting hosts communicating over the public Internet from targeted network attacks while preserving user privacy. To achieve this, repeated address rotation of the sender and receiver takes place. This address rotation can also occur in mid-session, preventing an attacker from knowing who the two hosts are. The MT6D IIDs are computed from a host's EUI-64 IID, a timestamp, and a shared session key. The authors ultimately validated their design with a proof-of-concept MT6D prototype.

[39] further adapted this approach for IoT. The authors investigated how this MT6D can be used with the "IPv6 over Low power Wireless Personal Area Network" (6LoWPAN). The authors concluded that frequent rotation of IPv6 addresses of 6LoWPAN devices can prevent an attacker from obtaining the IP address of a device and thus prevent an attack. Further research based on MT6D was done by [40], who presented optimizations and the design for a Micro-Moving Target IPv6 Defense (MT6D). This includes the protocols, the description of the operating modes, and the lightweight hash algorithms. Additionally, the authors present detailed testing and validation possibilities.

[41] also proposed MTD techniques for resource-constrained devices. Their approach involves reconfiguring such devices at two different architectural layers. The first is the security layer, where the reconfiguration is performed by switching between different cryptosystems in the embedded network. The second layer is the physical layer, where the reconfiguration of the devices can be done through different versions of the firmware. The authors concluded that their proposed mechanism increases the complexity for a potential attacker.

[42] suggested 6HOP. This algorithm provides transient addresses, ports, and key information for the connection endpoints (e.g. server and client). These endpoints must be initialized over a residential wireless network to exchange a secret. From this secret, the corresponding information is deterministically computed using the 6HOP algorithm. Thus, a server knows which address to assign to itself and on which port to listen for incoming requests, and the client knows where to connect.

[35], which was mentioned in the previous section, proposed several MTD mechanisms in addition to the MTD framework. The first mechanism is to honeypot and trap a crypto-ransomware encryptor with dynamically expanding and collapsing directories of dummy files. Along with this trapping, the encryptor is identified and killed. The second mechanism is to change the file extension of critical data to prevent it from being encrypted by crypto ransomware. This works because the file extensions determine whether or not the files are a target for some malware families. The third mechanism was to remove malware such as rootkits with a clean *ld.so.preload* file. The last mechanism was to randomize private IP addresses to deal with C&C malware, as the IP address change disrupts the communication between the IoT device and the C&C. These MTD mechanisms were then successfully tested against real malware.

[43] proposed an address shuffling algorithm (AShA). The idea of this algorithm is to let each node of a network calculate its new address autonomously. A coordinator ensures that each new address is not already in use in the network by choosing a set of parameters. ASha enables secure, fast, and collision-free address renewal in an IPv6 network.

[37] proposed two concrete MTD techniques using the framework explained above (IANVS). The first is a single port-hopping strategy using the UDP port number of a service as the moving parameter. Since known port numbers are a prerequisite for network services to work, their framework is required for port hopping while informing other parties in the network of the current port. The second strategy is to prevent DoS attacks on CoAP servers. The moving parameter here is the */.well-known/core* URI, and the idea is that the server should only respond to GET requests from clients if the encoded GET request matches the present MTD representation of */.well-known/core*. Finally, the authors evaluated the port-hopping strategy on real IoT hardware.

[36] proposed an MTD strategy that shuffles between 4 communication protocols between a node and the gateway in the IoT network by applying their proposed framework that was presented in the previous chapter. Examples of the communication protocols are WiFi and Bluetooth. By using the framework the authors found the ideal when to move parameter for their MTD strategy. This strategy ultimately involved changing the communication protocol with an uniform random shuffling and at a uniform random time between 1.5 minutes and 2.5 minutes.

Table 3.1 shows the described network techniques with some additional information. It is evident that there exist several different approaches to defend IoT against malware. However, no approach has been found that includes a cooperative component to more effectively defend against IoT malware. As there is an increasing threat caused by such malware [6] [27], further research is needed. This thesis addresses this research gap.

Source	Name of MTD Technique	Moving Parameter	Approach	Network type for which MTD was designed	Defend Against	Cooperative System
[39]	MT6D	IPv6 address	Hosts share symmetric key	WPAN	Network-side attacks (e.g. Denial-of-Service, Man-in-the-middle)	No
[40]	µMT6D	IPv6 address	Hosts share symmetric key	WPAN	Targeted network attacks	No
[41]		Cryptosystem	Change cryptosystem in security layer of device	Embedded networks		No
[42]	6HOP	IPv6 Address, ports and key information	Shared secret between 2 devices	Public Internet	Reconnaissance, address based correlation, DoS	No
[35]		Local IP address	MTD mechanism assigns new IP to device	Public Internet	C&C	No
[43]	AShA	IPv6 address	Shared key, PAN coordinator to prevent address collision in the PAN	WPAN	Various network attacks	No
[37]		UDP port	IANVS framework (Symmetric Key)	LAN	DoS	No
[37]		CoAP URL	IANVS framework (Symmetric Key)	LAN	DoS	No
[36]		Communication Protocol	Switch between 4 protocols		DoS	No

Table 3.1: A Table Summarizing Related Research.

Chapter 4

Implementation

This chapter presents the implementation details of the proposed solution, including the initial architecture created, the process of finding and modifying a suitable malware, and the explanation of the final implementation.

4.1 Creating the Implementation Environment

Before starting the implementation, a secure environment was essential, as this thesis deals with malware. To achieve this, Oracle VM VirtualBox Manager, an open source virtualisation solution [44], was installed to create three virtual machines. These included a leader machine and two other VMs called VM1 and VM2. While the leader machine is used to run all the server components, the other two are intended to mimic the IoT devices. Ubuntu was chosen as the operating system as it is the most widely used operating system for constrained devices as described in Section 2.2.3. This OS had to be changed later which will be described in Section 4.5.2. Since the hardware details can be customized later, no further thoughts were given to this at this stage.

Ideally, a different machine than the main computer would be used to run all the virtual machines, but unfortunately this was not an option. However, to ensure the highest level of security for the machine and the network, several security measures were taken. All the machines created were put into an internal network. This is an option within VirtualBox that creates a software-based internal network where only the machines in the network can communicate with each other. In this way, the machines cannot communicate with the host machine or the outside world, which was exactly the aim as this prevents uncontrolled spreading. A side effect was that it was not possible to download anything from the Internet.

The solution to this problem was a shared folder that could be accessed by the host computer and the virtual machines. For additional security, the shared clipboard has been disabled and the host machine has been placed on a guest network so that there are no other devices in the same network. Therefore, even in the unlikely case of the host machine being infected, the malware could not spread across the network except to the

router. However, as the default password has been changed and the latest updates are automatically installed on the router, the risk of infection is negligible. Other security measures that were already in place included a different OS on my host computer and Malwarebytes anti-virus protection. Thus, the environment created was as secure as it could be. The described environment can be seen in Figure 4.1.

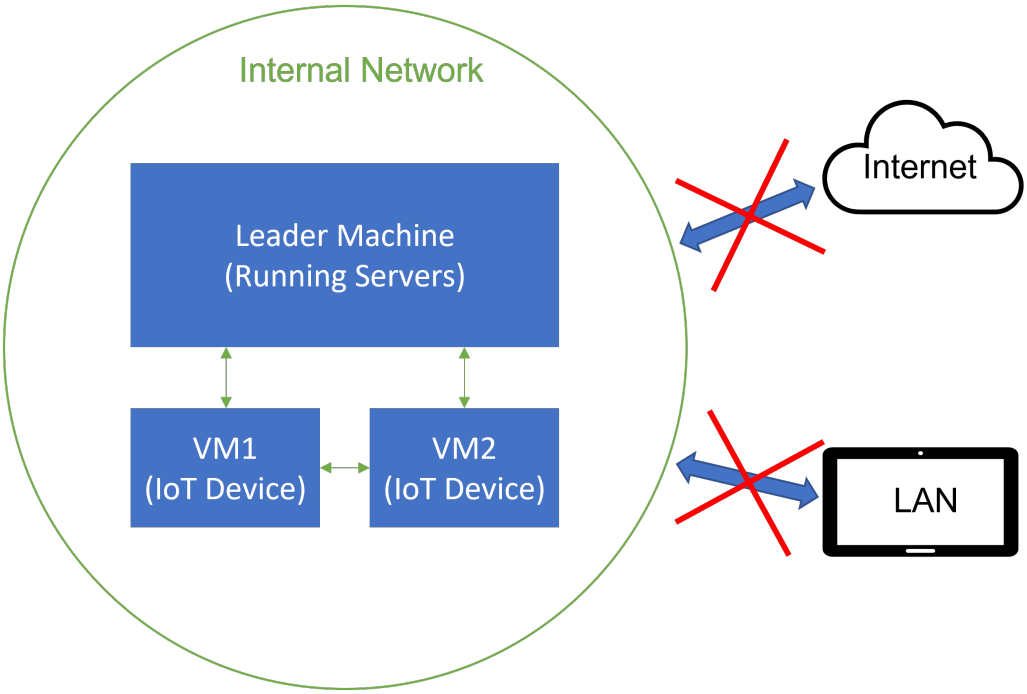


Figure 4.1: The Created Environment With 3 Different VMs in an Internal Network Which Prevents the Malware From Spreading to Other Machines.

4.2 Initial Prototype

The first prototype should help to better understand MTD from an implementation perspective, which should ultimately help to find appropriate solutions for the problem at hand. The architecture of this prototype was very similar to [10]. An overview of this architecture can be seen in Figure 4.2. The leader machine runs two applications, the *MTD Deployer Client* and the *MTD Deployer Server*. These two do not necessarily have to run on the same machine, but it is possible. The idea was to have these two programs do as much of the work as possible, given the hardware limitations of the IoT devices, at a later stage. The architecture is explained using the numbers in Figure 4.2.

1. VM1 and VM2 continuously send information to the *MTD Deployer Client*. This information consists of RAM usage and CPU usage.

2. The *Deployer Client* continuously checks for anomalies in the information sent by VM1 and VM2. For simplicity, this anomaly is a RAM or CPU usage greater than 50% in the last ten steps of the information sent.

3. The *Deployer Client* notifies the *Deployer Server* of conspicuous behaviour in VM1 and VM2.

4. The *MTD Deployer Server* then initiates the MTD techniques, which in this case is simply changing the private IP addresses of the virtual machines. The *Deployer Server* uses the *nmap* command to find all the occupied IP addresses in the network and sends each machine a new, unused IP address.

5. The VMs listen for commands sent by the *Deployer Server* and then migrate to the newly sent IP address using the *ifconfig* command.

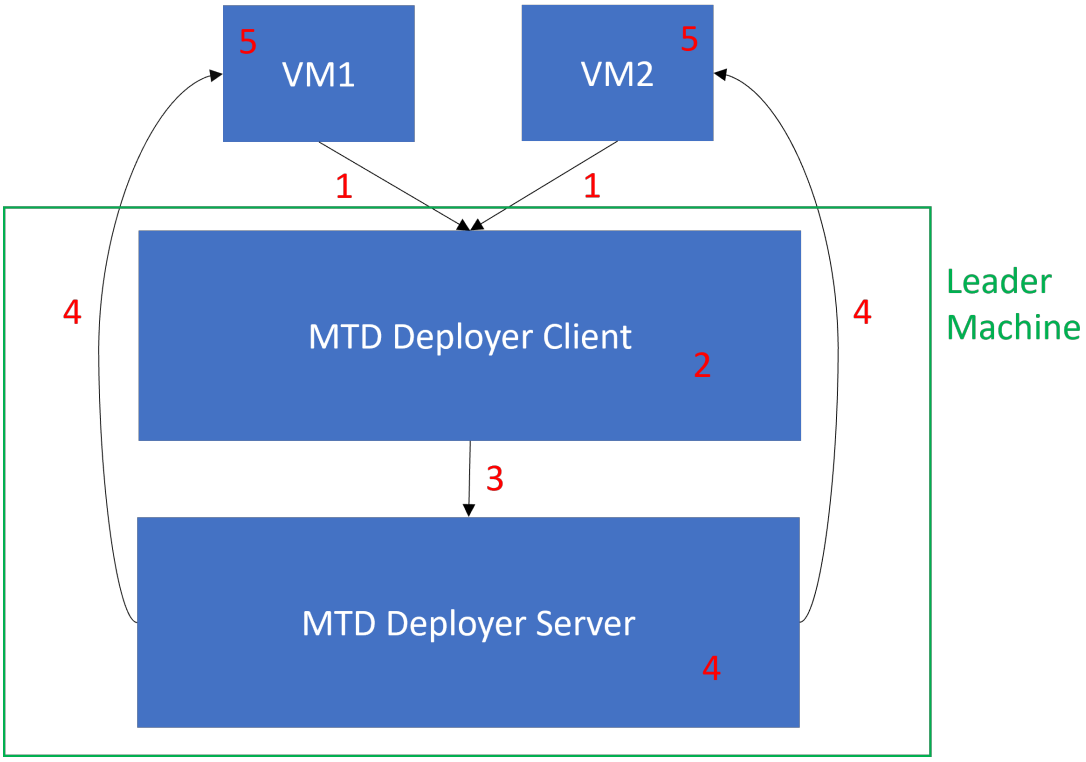


Figure 4.2: The Architecture of the First Prototype.

The cooperative part of this prototype is that if one VM detects malware, all VMs should initiate countermeasures, not just the affected one. This approach had several weaknesses, such as the VMs not checking that the migration to the IP address was working. This and other weaknesses were addressed in later implementations.

This prototype was implemented in Python using sockets, which can be used to send messages across a network [45]. The prototype was also briefly tested by running all the Python scripts and manually setting the CPU usage of a VM to 100%, whereupon the system handled the fake malware call. The *Deployer Server* successfully sent messages to the VMs, which then changed their IP addresses according to the content of the messages. Thus, everything worked as expected and it was possible to continue.

4.3 Finding Malware Samples

The first goal after the initial prototype was to find a suitable malware for this thesis. This was necessary for a number of reasons: Firstly, the malware had to be studied to see how it interacts with the system in order to develop the most appropriate countermeasures. This included studying the underlying code where possible. Second, it would be more meaningful to evaluate the MTD techniques with real malware. However, finding suitable malware proved to be a difficult and extremely time-consuming task. On the one hand, this was due to the rarity of malware code on platforms such as GitHub, which is understandable, as otherwise anyone could easily compile and distribute the malware. On the other hand, there were a number of requirements that the malware had to meet:

- The malware should ideally target IoT devices. Although it would have been possible to use malware targeting desktop devices, the aim was to use IoT malware to get as close to the real world as possible.
- The malware code should ideally be openly available for study and modification. This was important as the malware will likely need to be tailored to work for this thesis. It was also important to study the code in order to find possible weaknesses in the malware.
- The malware had to run on the created VMs in order to execute and play with it. This was ultimately to help understand and test the weaknesses of the malware.
- The malware must have a spreading functionality. This is necessary because the goal of this thesis is to use a cooperative MTD mechanism to mitigate/prevent the spreading of the malware.

Especially the last point proved to be difficult in the end. Several code repositories such as GitHub and malware repositories such as MalwareBazaar were searched, but no suitable malware could be found. On GitHub, there exists a repository [46] that contains the code of several IoT malware, such as Bashlite, Mirai, Lightaidra and some others. However, all of them had problems (incomplete, no spreading functionality, not executable) or were too complex to work with, as they might need to be slightly rewritten. There were other repositories, such as [47], which again provided the Mirai code, or [48], which provided the same code as the first repository. The problem with MalwareBazaar was that it provided mostly executables, which was unsuitable, since code to read and modify was an essential requirement. Finally, Bashlite was chosen, even though it was initially not executable and lacked an important piece of code.

4.4 Bashlite

This section presents various aspects of Bashlite. These include the changes required to get Bashlite running, explanations of how the malware works, and an analysis of where possible MTD techniques could be applied. Note that from now on the term ”client” will

be used to refer to the machines that act as infected and susceptible IoT devices. This is not to be confused with the *MTD Deployer Client* that runs on the leader machine and is part of the MTD framework.

The starting point was the code from [48], even though every Bashlite on Github seemed to provide the same code. Bashlite consists of two files, a server file and a client file, both written in C. Having encountered the C programming language only once in a university course and never used it again, the C language was the first obstacle, especially as the Bashlite code contained few explanatory comments. Once the general idea of the code was clear, the goal was to run it on the created VMs to see if this would work. The C files were compiled using the GNU compiler collection, which contains compilers and development tools [49] and was pre-installed on the Ubuntu systems.

Unfortunately, the execution completely failed in the case of the client script. The *server.c* file was executable on the leader machine, but the client threw a fork error that was unknown to me until then. *Fork* is used to create a child process, whereupon the parent and child process use separate memory spaces [50]. Although several possible solutions, such as running it with *sudo* or checking that the system had not reached its maximum number of processes were tried, the problem remained. After some time and a lot of trial and error, the idea came up that the underlying operating system might be the problem and not the script itself.

Thus, new machines were set up, but this time with the Raspberry Pi Desktop OS [51]. As shown in Section 2.3, the Raspberry Pi OS (formerly Raspbian) had the second-largest share of all Linux distributions in IoT devices, so it is a suitable alternative. After everything was set up, the client was immediately executable on the Raspberry Pi OS. Therefore, it is not clear whether the OS was the problem in the first place, but changing it was one possible solution. Below follows a description of Bashlite's server and client and how the two files were modified for this thesis.

4.4.1 Bashlite Server

The Bashlite server basically opens two ports on the executing machine. The first port (8888) was already defined in the code and is the port to which a "management" connects. This management then controls the server and the clients. So, to broadcast to the clients and do anything in general, one must first telnet to port 8888 of the server machine and enter a password, otherwise the server will not do anything useful with the clients. The other port opened by the server is passed as an argument when the script is started, along with the number of threads to create. The port used in this thesis was 6667, but this does not matter as long as the port is free; all that matters is that the same port is included in the client code, as this is the port to which the clients will connect after infection. The bridge between the management and the client is the broadcast function. This function takes the strings sent by the management and broadcasts them to all the clients, which then execute some commands/functions based on those strings. The important thing to note here is that no plain Bash commands can be sent directly from the management to the clients, so sending an "ls" as the management will not cause the clients to do anything.

The server automatically broadcasts a ”PING” to all clients every 60 seconds. Figure 4.3 shows the basic control flow of the server.

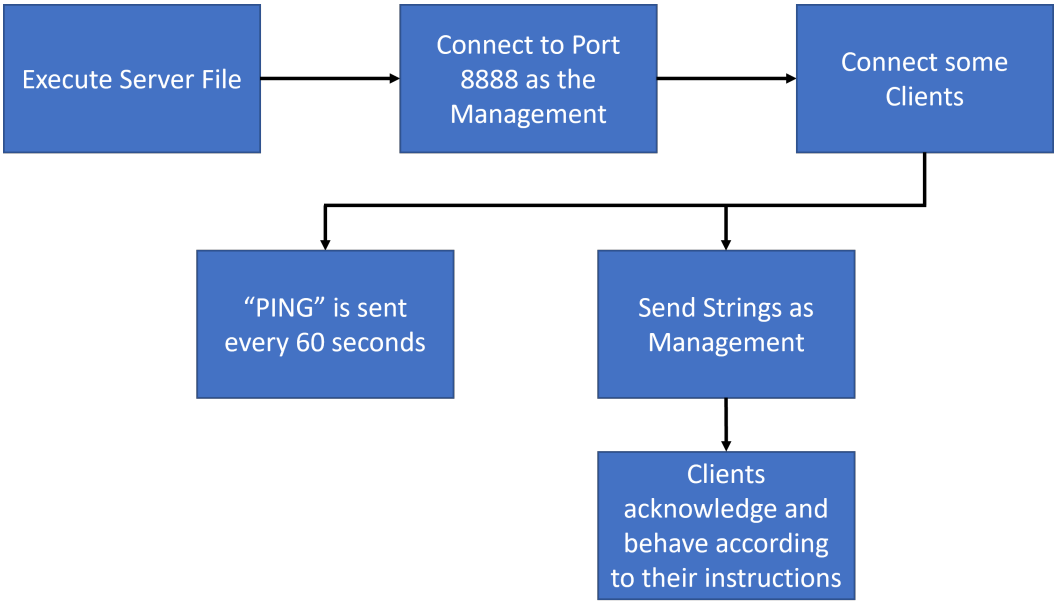


Figure 4.3: A Simplified Control Flow of the Server File of Bashlite.

4.4.2 Bashlite Client

The client is much more complex than the server and contains many more lines of code. At the beginning of the client file, the IP address of the command server must be specified, and two *char* arrays containing default usernames and passwords are created. Later in the code, these are used to check whether another device found is not properly secured, and if so, the correct username and password combination is stored. Various helper functions follow (e.g. various printing options, or a function that searches for a string in the buffer), but these are not important for a rough understanding of the general concept.

One important helper function is *getRandomPublicIP()*. This function first generates a random IP address using the *rand()* function, and then has a while block that basically checks if the IP address is public, and otherwise generates a new IP that is checked again, and so on. An invalid example would be if the first two octet portions of the IP address were 192.168.x.x, as this is a range for private IP addresses. Ultimately, the function returns a valid public IP address.

There are two other essential functions in the client, the *processCmd()* function and the *StartTheLelz()* function. The former processes strings sent by the management. The function waits for strings, whereupon the client returns a string or starts the execution of a function. Interestingly, some management strings are handled in this *processCMD()* function (e.g. SCANNER commands) and some strings are handled directly in the client's *main()* function (e.g. ”DUP”). Some commands have to start with an exclamation mark, otherwise they would not be recognized by the client. Useful inputs were

- ”DUP”, which terminates all client sessions
- ”!SCANNER ON” and ”!SCANNER OFF”, which starts and stops the *StartTheLelz()* function.
- ”!SH” *arguments* which can be used to run shell commands on the clients.

The *StartTheLelz()* function is the most important and also the most complex function of the client. At the beginning of this function a struct called *telstate_t* is created. This struct contains several attributes such as an IP address, the state, the username index or the password index and is inserted into a file descriptor array (fds[]). The state attribute is essential as there are 12 different cases in this function and the state attribute determines which case the current file descriptor is in. The function starts with case 0 and then increments to the next case if all requirements in the current case are met. The following paragraph briefly describes the main tasks of each case.

Case 0 checks if it is theoretically possible to connect to an IP returned by *getRandomPublicIP()*, and also increments the index of the username and password if the current file descriptor was sent back by Case 3 or 5. Case 1 checks for timeouts and the like. Case 2 checks if a login is requested by the connected IP address, Case 3 sends the username and also resets the file descriptor to Case 0 if the username was wrong, otherwise it sets the state to 4. Case 4 checks if a password is requested, Case 5 sends the current password of the file descriptor, and Case 6 checks if the password is correct and resets the file descriptor to Case 0 if not. Case 7 checks if the shell is accessible and Case 8 checks if Busybox is installed. Case 9 either sends a report to the server or continues with the subsequent cases which were not needed. The simple report sent to the server is just a string in the format ”REPORT IPaddress:username:password”. Figure 4.4 shows the process of the client in a simplified flowchart.

4.4.3 Code Modifications

In order to get Bashlite to work as it is supposed to, several parts of the code had to be modified. This subsection describes those changes. The client was simpler to modify, probably because the code was intended to run as it was from the start. In addition to many print statements to make the code understandable, the first step was to adapt the IP address of the command server and the arrays of usernames and passwords. Since the username and password of the susceptible machine are known, the corresponding combination was put into these arrays and the rest was deleted. Additionally, the script initializes another variable called ”stillRunLelz”. This variable was necessary because when the scanner on the first machine resumed scanning after sending the report to the server, the server often had trouble recognising that a second machine was now connected. So a workaround was to run the outermost while loop of the *startTheLelz()* function for as long as ”stillRunLelz” was 1, and after another vulnerable machine was sent to the server, ”stillRunLelz” was set to 0 and the scanner stopped. This would certainly not be an applicable solution in a real malware, but it was sufficient for this thesis, as one additional infected machine was enough to test the concept.

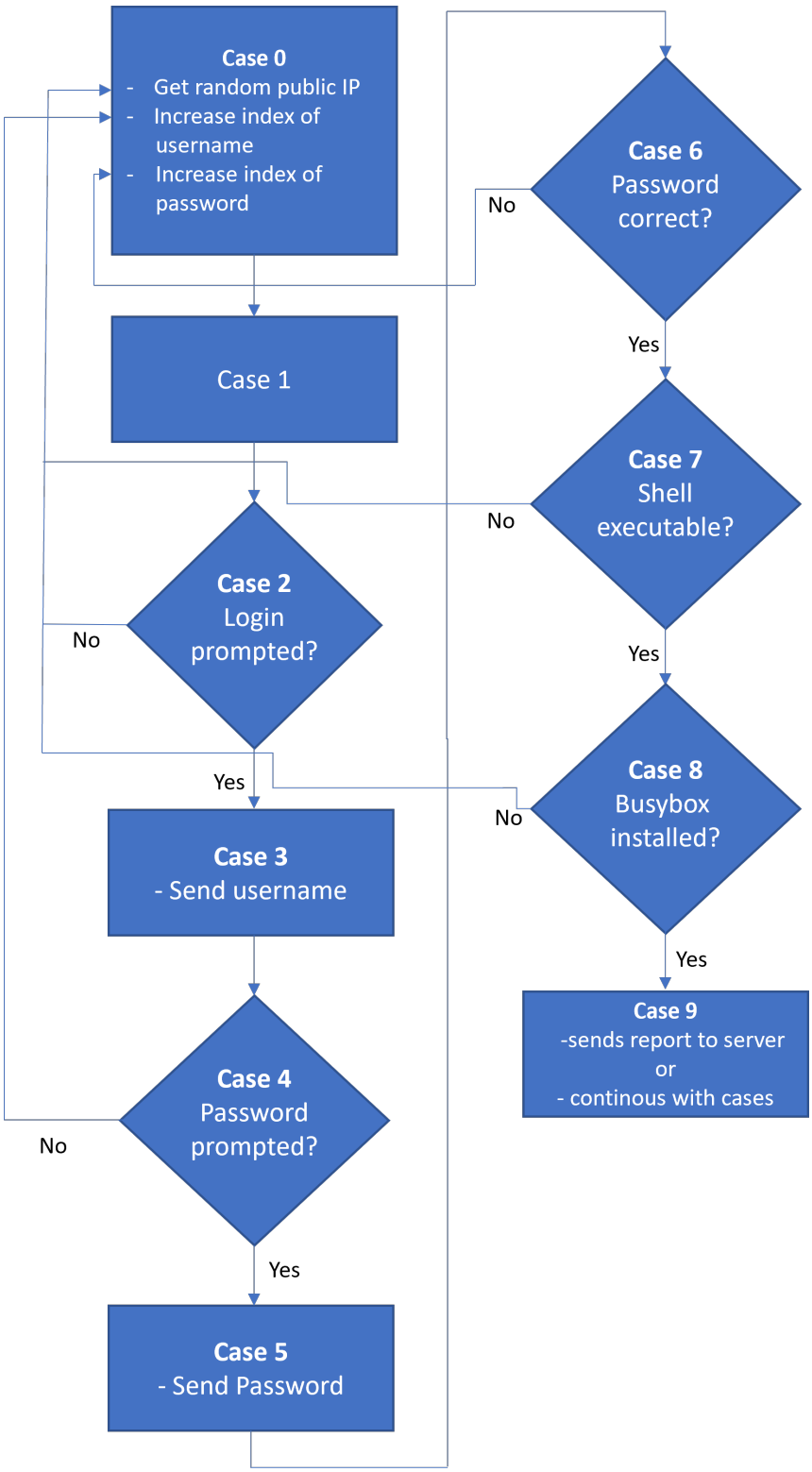


Figure 4.4: A Simplified Flow Chart of the Bashlite Client's Essential *startTheLelz()* Function, Which Searches for Other Susceptible Devices. The Incoming Arrows to Case 0 Directly Point to the Action That Will be Executed.

The *getRandomPublicIP()* function was also modified to return the IP address of the susceptible machine directly. Again, this would not make sense in real malware, but it sped up the development process without compromising the quality. Another problem was that the code never got further than case 6. This was solved by investigating where it failed and then sending it directly to case 7. Case 9 also checked a condition that was somehow not met, but this was solved by ignoring the condition and simply sending the report to the server at the beginning of case 9. These were all changes that had to be made to get the client to work so that a report of the vulnerable machine was sent to the server.

The server was more complex to adapt because some of the code was missing. Again, the first step was to add print statements to get a deeper understanding of how the server works. The second step was to add the missing code. This involved code to automatically exploit the IP address with the information sent by the client's scanner. To develop this, a separate C file was created so that Bashlite would not have to be continuously restarted. The code in this new file splits the report string into substrings to separate the IP address, username and password. The C function *strtok()* with ":" as delimiter helped to achieve this. The rest was more complex, as the server still had to automatically connect to the susceptible machine, copy the client file to that machine, and run the client file. The best way to do this was with a Bash script called from the *server.c* file with the *system* call. The first attempt of a Bash script using Telnet for the connection failed, probably due to the back and forth interaction between the two machines. The next attempt was an *expect* script. These scripts talk to other programs, know what to expect from them and give them the corresponding response [52].

The *expect* script starts with a timeout command of 30 seconds, so that it would not run for too long if it somehow failed. After that, the script creates variables for the arguments that are passed when the script is called. The call must be made in the following way:

telnetConnection.except *IPAddress username password fileToSend*

The first three arguments are given by the report string, the last one was a convenience in case a different file was to be sent to the client. The next step was to spawn the telnet command with the *IPAddress* argument, whereupon the script expected the string "raspberry login:" and then sent the *username* argument. This pattern continued until the *expect* script connected to the susceptible machine via Telnet. The script then needed to copy the client file from the server machine to the susceptible machine. This copying functionality was important because it could have provided another defence option at a later stage. Since installing additional packages or creating an ssh key or password was not an option, the options to transfer the file were limited. This brought *netcat* into focus, as it provided a good way to transfer the client file [53].

Thus, the susceptible machine should open a *netcat* connection on a port and then wait for the server to send the file over that connection. This worked perfectly when typed directly into Bash from both the client and the server, but it became more complicated than anticipated in the *expect* script due to two constraints. The first was that the client obviously had to open a connection before anything could be sent from the server, and

the second was that it was somehow impossible to send the files from the *expect* script on the local machine.

There were considerations to take the sending of the files out of the *expect* script (and have it done by another script), but this was not practical. The reason for this was that the *expect* script would then have to run partially (until it opened the *netcat* connection on the susceptible machine), then pause until the client file was sent by another script, and then continue running the rest of the *expect* script. The solution to this problem was to swap the server and client *netcat* tasks. Previously, the vulnerable machine would open a connection and wait for the file to be sent from the server machine. Now the server machine opens a port to send the corresponding file and waits for the susceptible machine to request it. This made it possible to first run the *netcat* command on the server and then start the *expect* script in the same *system()* call in C as the *expect* script requested the file from the server. Additionally, the content of the client file had to be pipped into *netcat* on the server side, otherwise the transfer would fail. The final command can be seen in Algorithm 4.1.

Algorithm 4.1 The Bash Command Used to Open a Netcat Connection From the Server to Send the File Once the Client Requested it and to Start the *Expect* Script

```
1 cat fileToSend | netcat -q 5 -l 9899 & expect telnetConnection.
    expect IPAddress username password
```

Algorithm 4.2 shows the *expect* script in pseudocode. Now everything worked and experiments with MTD techniques and malware spreading were possible.

Algorithm 4.2 The *Expect* Script in Pseudocode

```
1  SET timeout 30
2  SET IPaddress to argument 0
3  SET username to argument 1
4  SET password to argument 2
5  SET fileToSend to argument 3
6  CONNECT to IPaddress with telnet
7  EXPECT "raspberry login"
8  SEND username variable
9  EXPECT "password"
10 SEND password variable
11 SEND netcat command to request the file
12 SEND sleep for 7 seconds
13 SEND command to make the requested file executable
14 EXPECT "password for"
15 SEND password variable
16 SEND sleep for 3 seconds
17 SEND execute client file
18 EXIT expect script
```

4.4.4 Analysis of Bashlite

This section presents an analysis of Bashlite. This includes possible levers where MTD techniques could be applied to mitigate Bashlite.

As part of the analysis of Bashlite, many timing properties had to be checked. This required selecting suitable hardware properties for the VMs as a first step. As described in Section 4.1, the hardware details were not a concern initially. To simulate a Raspberry Pi device, hardware similar to that used by [35] was used, consisting of a 1.5 GHz CPU and 3.7 GB of RAM. The device running the VMs is a Surface Book 1 with 16 GB of RAM and an Intel Core i7-6600U dual-core CPU running at between 2.60 GHz and 3.40 GHz [54]. In VirtualBox it is possible to specify how many cores a VM can use and also the execution cap for those cores. Unfortunately, it is not possible to simply give the VM a frequency for its processor (e.g. 1.5 GHz). Thus, the performance information provided by the Windwows task manager had to be used as a reference to calculate the required execution cap. The task manager indicated that the laptop's base speed is 2.81 GHz. This appeared to be correct on average (even with all VMs running). Thus, each infected VM was given one core and an execution cap of 53%, because 53% of 2.81 GHz is about 1.5 GHz. In addition, the virtual machines have 3.7 GB RAM allocated to them by VirtualBox.

An infection process of Bashlite can be divided into four different phases, which are presented below.

1. The first phase is when the infected client scans for susceptible devices and sends the information to the Bashlite server.
2. The second phase is when the server receives the report from an infected device and initializes the Telnet connection to the susceptible client. The duration of this phase is extremely short.
3. The third phase begins as soon as the Telnet connection to a susceptible device is started.
4. The fourth phase begins when the Bashlite client is launched on the susceptible machine.

The first important data to get was the number of seconds these Bashlite phases took to complete. It is important to note that this could be different for a more sophisticated/varied malware or much better hardware on the devices, but Bashlite could serve as a reference. To achieve this, Bashlite was run 10 times and the time was measured by hand. Due to the various scripts on the server side, it was too complex to measure the time programmatically, so it was faster to do it manually. Other reasons were that it was not necessary to know the exact times and that the phases always took about the same number of seconds. The only time a script measured the execution time was in phase 2, as this period is too short to measure by hand. For phase 1, all time values were between 14 and 17 seconds, with an average of 15.5 seconds. Phase 2 lasted an average of 0.00007 seconds, making it negligible. Phase 3 time values ranged from 60 to 63 seconds with an

average of 62 seconds. Thus, on average, 62 seconds pass before the susceptible machine is infected. The long duration of phase 3 is due to the *expect* script written to connect, fetch the client file from the Bashlite server and execute it. This *expect* script needed several sleeps to complete the actions, and the Telnet connection also takes some time. Perhaps the whole script could be written differently, but this was not possible without putting too much effort in it. Figure 4.5 shows the corresponding timeline, where phase 2 and 3 have been combined and the phases are colour-coded according to whether or not it is still possible to mitigate the spreading to the second device.

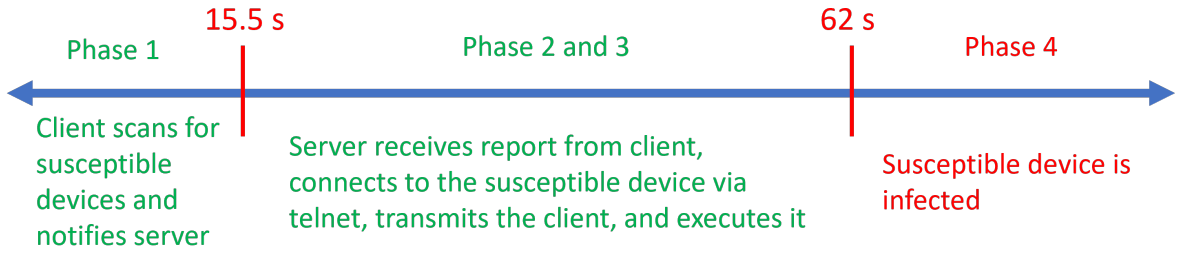


Figure 4.5: The Timeline Showing the Different Phases of a Bashlite Infection of one Susceptible Device.

4.5 Final Implementation

This chapter presents the final implementation based on the analysis of Bashlite in Section 4.4.4. The implementation is based on the same structure in terms of VMs as the initial prototype in Section 4.2. Again, the structure consists of three different virtual machines. The first is a leader machine running two applications, the *MTD Deployer Client* and the *MTD Deployer Server*. Both applications consist of two Python files, one of which acts as a listening socket, while the other contains the essential helper functions for the listening socket. For example, the *MTD Deployer Server* has a listening Python file called *listenToDeployerClient.py* and a *sendToDevices.py* that sends the MTD execution commands to the devices.

The other two machines are the clients that mimic the IoT devices and also consist of two files each. The *sendToDeployerClient.py* notifies the *Deployer Client* when malware has been found, and the *listenToDeployerServer.py* executes the corresponding MTD technique using the information sent by the *sendToDevices.py*. Even though this chapter briefly explains every file, the focus lies on *sendToDevices.py* (*Deployer Server*) and *listenToDeployerServer.py* (client), as these two form the basis of the MTD techniques. However, before explaining the code, this section first introduces the chosen MTD techniques and how they relate to the background information from Section 2.

4.5.1 Implemented MTD Techniques

The analysis of Bashlite described in Section 4.4.4, together with the working implementation of Bashlite described in Section 4.3, provided the opportunity to search for suitable

MTD techniques. These techniques should ideally protect against classic botnets and P2P IoT botnets. The first subsection below describes the implemented MTD techniques based on the analysis of Bashlite in Section 4.4.4, and the second subsection theoretically evaluates how these MTD techniques against Bashlite could protect against Mirai and HEH.

Implemented MTD Techniques Based on the Analysis of Bashlite

The first phase of Bashlite ends with the report sent from the infected client to the Bashlite server. [35] and further experiments in this thesis have shown that it is possible to disrupt the connection between the Bashlite client and the Bashlite server by changing the IP address of the client. If the connection is disrupted before phase 2 begins, the report can be prevented from being sent, resulting in the best possible mitigation case. Once the second phase begins, the IP address change of the originally infected machine (VM1) will have no effect on the spreading of Bashlite to the susceptible machine (VM2). Of course, the IP address change should still be executed to disconnect the infected machine from the server, but this has no advantage in terms of spreading to the susceptible machine (VM2). This makes the second technique all the more important. The idea of this second technique is to move the Telnet port of the susceptible machines for some time while Bashlite is rendered harmless on the infected machine. This essentially hides the susceptible machines from the infected machine's scanner.

This Telnet service port change technique works in phase 1. As just described, this prevents the infected machine's scanner from finding the susceptible machines. This technique theoretically also works in phase 2, although it is unlikely that the Telnet service port change is applied at this point because phase 2 is so short. The port change also works in phase 3, but it requires a small adjustment because once the connection between a susceptible machine and the Bashlite server is established, the port change has no effect. Thus, a potential Telnet connection must first be killed, and then the port changes must be applied to ensure that this susceptible machine is completely unreachable to a Bashlite scanner. In phase 4, the second machine is already infected, but the Telnet service port change should still be applied to all other machines in the network, as it could protect other, not yet infected machines (which were not present in the setup). Figure 4.6 graphically shows the MTD techniques in combination with the phases. Note that the colours only indicate whether the applied MTD technique can prevent Bashlite from spreading to the susceptible machine. So a red font does not mean that the technique should not be applied, but that it has no effect on the Bashlite infection of other machines in the network. A green font means that the Bashlite spreading can be completely mitigated, a yellow font means that the spreading to the susceptible machine could not be mitigated, but the port change could still help other machines in the network.

Theoretical Evaluation of MTD Techniques Against Mirai and HEH

A quick examination of Mirai's scanner file allowed an assessment of whether the proposed MTD techniques would work against Mirai without running it. Although there is a big

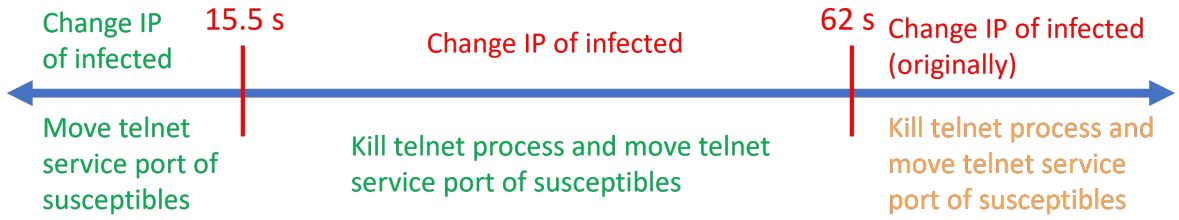


Figure 4.6: The Timeline of an Infection With Bashlite of a Susceptible Device, Showing When the Mitigation is Still Possible.

difference in the code, it is obvious that Mirai's scanner also targets Telnet ports, specifically port 23 and port 2323. The latter is often used as an alternative for port 23 [55]. The same holds for the HEH malware presented in Section 2.3.2, as this malware also uses port 23 and port 2323 to infect devices [29]. It is therefore safe to say that MTD techniques protecting port 23 (2323) should also work against Mirai and HEH, at least in theory. This is not the case for the IP address change. Although it is not possible to test it in the scope of this thesis, Mirai has a teardown functionality that is triggered when no response is received from the C&C server. After this teardown, the client simply reconnects to the C&C server, making the IP address change only a temporary obstacle. As for the HEH malware, it is not possible to predict the impact of the IP address change due to the unavailability of the code. However, as these types of botnets are extremely dangerous by nature, it is all the more important to prevent them from spreading. Changing the Telnet service port can therefore be considered an essential tool against all three malware types.

Once the required MTD techniques had been chosen, they needed to be defined according to the background information given in Section 2. This chapter introduced the three elements that define MTD techniques [13]. These three elements are "what", "how" and "when" and are used to formally describe the solution below.

Formal Definition of the Applied MTD Techniques

The "what" to move is the IP address value and the port value of the Telnet service. The domain from which these two values are taken is defined in a configuration file, so that the user can choose in which range the value of the moving parameter should be. Of course, any IP address or port already occupied in the network or on the machine is automatically removed from this domain. The "how" is completely random. A random IP address or port value that is within the specified range but not yet in use will be used to replace the current moving parameter value. The final element is the "when" to move. As Bashlite has been shown to be detectable [35], the proposed solution uses an event-based decision process that includes a proactive and a reactive component. The IP address change technique is the reactive component, as it reacts to the detection of Bashlite and then initiates the countermeasures. In a sense, the port change is both reactive and proactive. Although it is also triggered by the detection of Bashlite (reactive), it also acts as a proactive component on the uninfected machines, as the technique attempts to get ahead of a Bashlite infection. The information just described can be seen graphically in Table 4.1.

MTD Techniques	What		When		How	MTD Domain
	MP Value	MP Domain	Decision Process	Active vs. Reactive		
IP Address Change	IP address value	Given in config minus occupied	Event-based	Reactive	Random	Dynamic Networks
Telnet Service Port Change	Telnet service port value	Given in config minus occupied	Event-based	Both	Random	Dynamic Networks

Table 4.1: A Summary of the MTD Techniques Applied With Respect to the Three Fundamental Elements of IoT Techniques.

Although both MTD techniques clearly fall into the dynamic network category, the classification of the attack phase they seek to disrupt is not as straightforward as [8] indicated. The reason for this is that I consider the five-phase attack of [8] to be more applicable when an attacker has a specific target to attack. Bashlite aims to infect as many generic devices as possible, rather than targeting a specific device. However, Bashlite also has its different phases, as shown in Section 4.4.4. This allows a comparison between the attack phases specified by [8] and the phases of Bashlite. The scanning phase of Bashlite corresponds to the Reconnaissance and Access phases, as they both attempt to find a target and gather information about it. There is no Bashlite counterpart to the Development phase, as the malware is obviously already developed. The transfer of the client file to the susceptible machine and its execution corresponds to the Launch phase. Although Bashlite does not install any additional backdoors, the fact that the client is connected to a C&C server could be mapped to the Persistence phase of [8]. The information just described can be seen in Table 4.2.

Since Section 4.4.4 showed in which phases of Bashlite it is possible to mitigate its infection, this can also be expressed in terms of the attack phases of [8]. Hence, the MTD techniques applied defend against the Reconnaissance phase, the Access phase and the Launch phase. [8] also indicated that Dynamic Network Domain techniques can hinder the Reconnaissance and Launch phases of an attack, but they did not indicate that Dynamic Network Domain techniques can also hinder the Access phase. I assume this is due to the difference in the type of attack (Bashlite vs. specific target).

Bashlite Phases	Attack Phases				
	Reconnaissance	Access	Development	Launch	Persistence
Phase 1 (Scanning)	x	x			
Phase 2 and 3 (Transfer and execution of client)				x	
Phase 4 (Connected to command server)					x

Table 4.2: The Attack Phases of Bashlite and Their Mapping to the Corresponding Attack Phases.

4.5.2 Issues of the Initial Prototype

The first step towards the final implementation was to test whether the initial prototype would still run on the current machines. Unfortunately, this was not the case, as the *ifconfig* command used to change the IP address in the initial prototype in Ubuntu was not suitable for changing the IP address in Raspberry OS. Another problem with the initial implementation was that each machine's IP address was changed. Assuming that a machine is infected and Bashlite starts scanning for other susceptible machines in the LAN, changing the IP address of the other machines would make no difference because Bashlite continues to scan for IP addresses randomly, meaning that another IP address would still be found. Furthermore, this thesis has not considered that the LAN setting could have other negative effects. Section 7.1 briefly discusses this issue, but it was beyond the scope of this thesis to address it.

4.5.3 Implementation Details

This chapter presents and explains the implemented code. It separates the clients, *Deployer Client* and *Deployer Server* as well as possible, but the main goal was to explain the files in the order of the control flow. A recurring Python module in the code is the *subprocess* module for running Bash commands in Python. Two different functions from this module were used, the *subprocess.run()* and the *subprocess.popen()*. A difference between the two is that the *run()* function waits until the Bash command has finished, whereas the *Popen()* executes the command in a child process [56]. In practice, this means that *subprocess.run()* will cause the Python code to pause its execution until the passed Bash command is complete, and *subprocess.open()* will continue to execute the Python code without waiting for the Bash code to complete. Depending on the Bash command and its goal, it was clear which function was beneficial.

Device Sends Information

The starting script is *sendToDeployerClient.py*. This script is quite simple and differs significantly from the initial prototype. First it connects to a listening socket created by the *DeployerClient*. This is followed by a try and except block in which the scripts calls a function that starts Bashlite using the *subprocess.Popen()* function. This ensures that Bashlite runs in the background and the Python script can continue. A modified version of Bashlite that immediately starts scanning for susceptible devices saved a lot of time, as there was no need to connect to the server as management and manually start the scanner over and over again. The *sendToDeployerClient.py* script then sleeps for a certain number of seconds. This was to fake the time it took to detect Bashlite, as after this sleep the script sends a "malware found" message to the *Deployer Client* listening socket.

Deployer Cient Handling Client Information

The listening socket of the *Deployer Client* is created by the *listenToDevices.py* script. Again, this is a fairly simple script that just listens for virus messages and calls the *send()* function of the *sendToDeployerServer.py* script. This listener supports multiple connections through multithreading. This was necessary because all devices in the LAN should be able to send information to the *Deployer Client* at the same time. In the *send()* function of *sendToDeployerServer.py*, the script connects to the listening socket of the *Deployer Server* and sends another virus message along with the IP address of the infected machine.

Deployer Server Handling Deployer Client Information

The *listenToDeployerClient.py* script creates this listening socket and receives the virus message from the *Deployer Client*. This message is then split to filter out the IP address, followed by a call to the main function of the *sendToDevices.py* script with the IP address as an argument. This IP address argument indicates the IP address of the infected device.

Before explaining the *sendToDevices.py* script, this paragraph briefly explains the *Config.json* file. This is where the user can add configuration values regarding the MTD techniques. The configuration file can be seen in Listing 4.1. There are several key-value pairs for each MTD technique (IP address and Telnet service port change). The rootIP, together with the startIPDevices and endIPDevices, determines the possible range of IP addresses that could be assigned to an infected device. So, in this listing example, the possible IP address of an infected device is in the range between 192.168.1.1 and 192.168.1.90 minus the IP addresses already in use. The server IP is generally used on multiple occasions throughout the code. As for the port change MTD technique, the "startOfPossible" and "endOfPossible" keys define the port range that could replace the Telnet service port on the susceptible devices. The "timeToChangeBack" key specifies the number of seconds that susceptible machines should wait before the Telnet service port is changed back to port 23. This information will eventually need to be sent to the client, but the aim was to have the entire configuration in one place.

Listing 4.1 The Configuration File in Which the MTD Values can be Specified.

```
1  {
2  "IP": {
3   "rootIP": "192.168.1",
4   "startIPDevices": "1",
5   "endIPDevices": "90",
6   "serverIP": "10"
7   },
8  "port": {
9   "startOfPossible": "3000",
10  "endOfPossible": "4000",
11  "timeToChangeBack": "300",
12 }}
```

The *sendToDevices.py* is one of the two essential files. This file contains two functions. The first is a helper function called *getIPInformation()*. This function takes a set of arguments specified in the Config.json file and returns an IP object containing all the relevant information regarding IP addresses. First, the function creates an array of all possible IP addresses within the range and the server IP address specified in the configuration. Then the function runs a *nmap* scan on that IP range to see which IP addresses are already in use. The function filters the result of the *nmap* with a regex expression, removes the occupied IP addresses from the array of all possible IP addresses, and creates an array in which it puts all occupied IP addresses except the server IP address and the infected IP address. This array is needed later and contains all the IP addresses that may need to move their Telnet service ports for the Telnet service. Finally, the object returned by the *getIPInformation()* function is a dictionary with three key-value pairs containing the IP address of the infected device, the array of all possible/unoccupied IP addresses, and the array of all IP addresses that may need to change their Telnet service ports for the Telnet service.

The other function in the *sendToDevices.py* is the *letExecuteMTDMechanisms()*. This function triggers the start of the MTD techniques on the clients and provides them with the necessary information. It needs the information object returned by the *getIPInformation()* function and other arguments from the configuration file. *letExecuteMTDMechanisms()* starts by assigning the values of the information object to variables to keep the code as clean as possible. This is followed by two main blocks of code, the first dealing with the IP address change of the infected machine and the second dealing with the port change.

The code dealing with the IP address change starts with an if statement to check if an IP address was passed as a parameter in the function call. This allows the user to only change the port if he/she has a use for this, and it allowed to better test the code as it is possible to run the port change MTD technique alone. The if statement is followed by a try and except block to catch any errors. The try block connects to the infected device's listener, picks a random IP address from the array of all possible IP addresses, and sends a message of the form "IP:*IPaddress*". But before this message is sent, a "-1" is sent to the listeners (clients) and the response is received. This was an improvised and slightly

unclean fix, but since the client expects one more round of information from the *MTD Deployer Server* in the port change code block than in the IP change code block, this allowed to leave the code as it was on the client side. After the IP message is sent, the port is closed. Again, several exceptions are caught to prevent unexpected code failures. The IP address change code block can be seen in Algorithm 4.3.

Algorithm 4.3 The Code Block for Changing the IP Address on the Server Side From the *letExecuteMTDMechanisms()* Function in Pseudocode. Everything Related to the Port Change has Been Ommited Here.

```
1  def letExecuteMTDMechanisms(ipInformation, timeToChangeBack,
       maxTryOfNewPorts, startOfPossiblePorts, endOfPossiblePorts):
2    // NOTE: omitted all the code for the port change
3
4    SET allPossibleIPS to ipInformation["allPossibleIPs"]
5    SET infectedIP to ipInformation["infectedIP"]
6    IF infectedIP is not None:
7      TRY:
8        CONNECT to client via socket
9        SET newIP to random IP of allPossibleIPS
10       SEND "-1" to client
11       RECEIVE unimportant answer from client
12       SEND "IP:newIP" to socket
13       CLOSE connection to client
14     CATCH ERROR:
15       PRINT some kind of error message
16   END IF
```

The code dealing with the port change starts with a for loop because, unlike the IP address change, the port needs to be changed on multiple devices. So the script iterates through all the IP addresses in the array of IP addresses that might need a port change. This is followed by another try and except block. Several things happen in this try block: a connection is made to the currently iterated IP address, an array is created containing all the ports defined as the port range in the configuration, and a random port is selected to be sent to the device. The script also sends the maximum number of attempts the client should make to migrate to another port. This is followed by two loops to continuously talk to the clients.

The inner while loop continuously sends a message ("port:*Port*") with the randomly selected port to the client, which checks if the port is still available and sends a corresponding response. If the port is available, the script exits the inner loop and continues its execution. If the port is not available on the client, the port is removed from the array of available ports and a new port is randomly selected and sent again. In addition to these two options, the script also handles cases where the maximum number of ports to look up has been reached, or the ports on the client cannot be looked up at all.

As soon as the code breaks out of the inner loop, the script does nothing, but waits for another message to be received. Here it checks three different cases: either the migration

to the new port worked, an error was received, or the client asks for the time after which it should move the telnet service back to port 23. In the first two cases the script returns from the function, in the third case it sends the value of the *timeToChangeBack* key from the configuration file. In the except block, the script catches a *ConnectionRefusedError*, because if a device that should change the port of the Telnet service does not have the listener running, the code would otherwise throw an error and break. The code block for changing the port on the server side can be seen in Algorithm 4.4.

Algorithm 4.4 The Code Block for Changing the Port of the Telnet Service on the Server Side From the *letExecuteMTDMechanisms()* Function in Pseudocode. Everything Related to the IP Address Change has Been Ommited Here.

```
1  def executeMTDMechanisms(ipInformation,timeToChangeBack,
       maxTryOfNewPorts,startOfPossiblePorts,endOfPossiblePorts):
2    // NOTE: omitted all the code for the IP change
3
4    SET IPsToChangePort to ipInformation["IPsToChangePort"]
5    FOR each IP in IPsToChangePort:
6      TRY:
7        CONNECT to IP of client via socket
8        SET allPossiblePorts to range in config file
9        SET newPort to random port of allPossiblePorts
10       SEND maxTryOfNewPorts given in configuration file
11       RECEIVE unimporTant answer from client
12       SET success to False
13       WHILE True:
14         WHILE success is False:
15           SEND "PORT:newPort" to socket
16           RECEIVE answer from client:
17           IF answer is that port is free on client:
18             SET success to True
19           ELSE IF answer is that telnet port not retrievable:
20             PRINT some kind of error message
21             CLOSE connection to client
22           ELSE IF answer is that max attempts are exceeded:
23             PRINT some kind of error message
24             CLOSE connection to client
25           ELSE:
26             REMOVE newPort from allPossiblePorts
27             SET newPort to random port of allPossiblePorts
28           ENDIF
29         END WHILE
30         SET answer to received data from client
31         IF answer is that client finished:
32           PRINT success message
33           CLOSE connection to client
34         ELSE IF answer is that error ocurred:
35           PRINT some kind of error message
36           CLOSE connection to client
```

```
37          ELSE IF answer is that client needs num of seconds:
38            SEND seconds to move back from configuration
39          ENDIF
40        END WHILE
41      CATCH ERROR:
42        PRINT some kind of error message
43    END FOR
```

Client Handling Deployer Server Information

The *listenToDeployerServer.py* is the most important script of the whole solution, as it actually executes the MTD techniques on the clients. This is the only script that needs to be run with *sudo*, as some of the Bash commands included require this security privilege. The script has four small helper functions and one large helper function. The four small functions include one that writes to a log file called "output.txt", one that queries the current port of the Telnet service on the machine, one that checks if the Telnet service is listening on port 23, and one that kills a potential Telnet process. The logging functionality was crucial for debugging reasons. It helped to detect errors more quickly, and it is also essential in a real-world deployment of the MTD techniques.

The second little helper function was to query the current port of the Telnet service. This was needed because the change back to port 23 was not implemented from the start. This prevented the need to manually reset to port 23 after each run during the implementation process. More importantly, the Telnet service port query also catches possible errors that might occur in general. The *getTelnetPort()* function first executes the Bash command shown in Algorithm 4.5.

Algorithm 4.5 The Bash Command Used to Query the Telnet Service Port.

```
1 ss -tlpHn | grep inetutils-inetd
```

The *ss* command in Algorithm 4.5 can be used to examine sockets, similar to netstat [57]. The options passed are: include tcp sockets, include listening sockets only, include processes, output information without the header, and try not to resolve service names [57]. Additionally, the code filters with *grep* for "inetutils-inetd", which was used as an identifier for the Telnet process. *Inetd* is a program that listens on certain Internet sockets and decides which program should respond to the request [58]. Using this as an identifier works perfectly on the VMs because Telnet is the only service on the VMs that uses *inetutils-inetd*. As this may be different on other devices, it is possible that this Telnet port query may fail. To ensure that the script does not query the wrong port for the Telnet service and therefore misbehaves, the script also queries the Telnet service port from the */etc/services* file and compares the two. If they are identical, all is good, otherwise the script throws an *AttributeError*. The reason for the *AttributeError* is that it is needed anyway in case "ss -tlnpH | grep inetutils-inetd" returns nothing, which would lead to an *AttributeError* caused by the regex the script uses to filter the port.

The third small but important helper function is the *checkIfTelnetWorks()* function. This uses a socket to check if the Telnet service port allows a connection or not. This function is used to check if moving the Telnet service port was successful or not.

The fourth helper function is *killTelnetProcess()*, which does exactly what it says, again using the *subprocess.run()* function to execute the Bash command. The Bash command passed is a *pkill* with an echo, which echoes either "True" or "False" depending on whether *pkill* was successful or not, and logs accordingly.

The fifth and larger helper function is the most complex and is called *changePort()*. It again starts with a command passed as an argument to the *subprocess.run()* function. The command can be seen in Algorithm 4.6.

Algorithm 4.6 The Bash Commands to Replace the old Telnet Service Port With the new one and Apply the Change.

```
1 sed -i 's/\<{0}\>/{1}/g' /etc/services
2 sudo systemctl restart inetutils-inet.service
```

The {0} and {1} in Algorithm 4.6 are placeholders which are inserted using Python's *string.format()* function. The first line looks for the old Telnet service port number plus "/tcp" (e.g. 23/tcp) in */etc/services* and replaces it with the new port number plus "/tcp" (e.g. 2255/tcp). Initially, the mistake was made of not looking for the exact string, which led to other ports containing parts of the old port (e.g. 5523/tcp) being changed (e.g. to 552255/tcp). The solution was to surround the first placeholder with "<" and ">", which resulted in an exact search for the value to replace [59].

The second line of Algorithm 4.6 restarts the *inetutils-inet* service, which applies the changes in the */etc/services* file without rebooting the system. The script then uses the *grep* command to check if the new port can be found in */etc/services* and echoes accordingly. Depending on the echo, the script logs an error, returns from the function, or continues with the code. Next, the script uses the *checkIfTelnetWorks()* function to see if the Telnet service is still listening on port 23. If it is, the script writes an error to the log file and sends an error message back to the *Deployer Server*. Otherwise, the code logs that the Telnet service is no longer listening on port 23 and sends a "done" to the *Deployer Server*. Additionally, the script triggers the Telnet service to change back to port 23 after a certain number of seconds. This number is also specified in the *Deployer Server* configuration file. To initiate this change back, the script uses the *subprocess.Popen()* function instead of the *subprocess.run()* function, as the Python script should continue to run. The command passed as an argument to *subprocess.Popen()* can be seen in Algorithm 4.7.

Algorithm 4.7 The Bash Commands to Change Back to the Telnet Service Port 23 Afer a Given Number of Seconds

```
1 printf "$(date +%F\ %H-%M-%S) SHELL: Sleeping for {0} seconds\n
    " >> output.txt
2 sleep {0}
```

```
3  sed -i 's/\<{1}\>/23\/tcp/g' /etc/services
4  sudo systemctl restart inetutils-inetd.service
5  sleep 7
6  sudo lsof -i:23 && found="true"
7  if [ "$found" = "true" ]
8  then
9    printf "$(date +%F\ %H-%M-%S) SHELL: Went from port {2} back
      to port 23\n\n\n" >> output.txt
10 else
11   printf "$(date +%F\ %H-%M-%S) SHELL: ERROR: The change back
      to port 23 failed somehow\n\n\n." >> output.txt
12 fi
```

The first two lines of Algorithm 4.7 write a log to the same file where the Python code is logged. Then the script sleeps for the specified number of seconds. Again, these numbers are inserted using Python's *string.format()* function. Lines 4 and 5 are almost identical to the *changePort()* function, except that the target port is now port 23 again. After modifying the */etc/services* file, the *inetutils-inted.service* is restarted to take effect without rebooting the machine. The script then pauses for 7 seconds to allow the changes to the port to take full effect before continuing. On line 7, the script runs a *lsof* command to check if port 23 is listening again. This is the Bash replacement for the *checkIfTelnetWorks()* function in Python. Ideally the script would check directly with Telnet if the connection is possible, but it was not possible to make this work with the *subprocess* module. If port 23 is listening, the script logs a success message to the log file (on lines 8 and 9), otherwise it logs an error to the log file (on lines 10 and 11).

The main function of the *listenToDeployerServer.py* script is *listenToDeployer()*. First the listening socket is created, followed by the first while loop. In this loop, the script receives the maximum number of attempts to look for a free Telnet port sent by the *Deployer Server*, which queries the configuration file for the maximum number of attempts. The script then sends a response that the maximum attempts have been received, which also ensures that the *Deployer Server* does not continue with its code as it is forced to wait for a response. A second loop follows in which the script receives another message from the *Deployer Server* that is either "IP:*IPaddress*" or "port:*Port*". This message is split into two variables called "movingParameter" and "movingParameterValue". The script then checks whether the moving parameter is "IP" or " port".

In the IP case, the script closes the listening socket properly, as it would have been closed anyway after the IP change, and all the necessary information has been received. It then kills the Bashlite process. Although the IP change disconnects the Bashlite client from the command server, rendering it harmless, the Bashlite client continues to run on the infected machine, which was tedious for testing. Therefore, the Bashlite client was killed. The IP address of the device is then changed using the command shown in Algorithm 4.8.

Algorithm 4.8 The Bash Commands to Change the IP Address.

```
1  sed -i 's/\<{0}\>/{1}/g' /etc/dhcpcd.conf
```

```
2 ifconfig eth0 down && sudo ifconfig eth0 up
```

The first line of Algorithm 4.8 replaces the old IP address with the new one in the *etc/dhcpcd.conf* file. The script again inserts the values in Python with the *string.format()* function, therefore the {0} and {1} in the Bash code. The second line shuts down the Ethernet adapter and restarts it immediately. The Raspberry OS needs to do this to migrate to the new IP address, otherwise the changes would not take effect until the system is rebooted, which is not applicable. After the IP address is changed, the script sleeps for 7 seconds because the Ethernet adapter needs about 5 seconds to reboot. After this time, the script uses the *socket.gethostbyname()* function to get the current IP address of the device, checks whether the IP address change worked or not, and logs a corresponding message. The Python code block for changing the IP address on the clients can be seen in Algorithm 4.9

Algorithm 4.9 The Code Block for Changing the IP Address of a Client From the *listenToDeployer()* Function in Pseudocode. Everything Related to the Port Change of the Telnet Service has Been Ommited Here.

```
1  def listenToDeployer(HOST, PORT):
2    CREATE listening socket
3    WHILE True:
4      RECEIVE max attempts for trying to find port (-1 here)
5      SEND unimportant answer
6      WHILE True:
7        RECEIVE command from server
8        SPLIT command to movingParameter and movingParameterValue
9        IF movingparamter is "IP":
10         CLOSE connection to client
11         KILL telnet process
12         CHECK if telnet process was killed
13         CHANGE the IP address
14         SLEEP for 7 seconds
15         CHECK if the IP change worked
16       // NOTE: omitted all the code for the port change
17       ELSE
18         CLOSE socket
19         RETURN
20       ENDIF
```

If the moving parameter is the port, then another machine in the network is infected and the current machine should move its Telnet service port. There is an additional condition in the port case if statement. This condition is whether a count variable is below the maximum number of port change attempts.

The port change code block is more complex than the IP change block above. In a first step, the script checks the */etc/services* file to see if the random port sent by the *Deployer Server* is free on this device. If the port is found in the services file, the script sends a

"taken" message to the *Deployer Server*, which then sends a new random port as described above. If this port is free on the device, the script will query the current Telnet port using the *getTelnetPort()* function described above. The script also calls the *killTelnetProcess()* function. This is necessary because, as mentioned above, if the Bashlite server has already started a connection to this susceptible machine, changing the port of the Telnet service would have no effect, and the Bashlite client would normally start on the susceptible machine.

Note that the *killTelnetProcess()* function only affects existing connections, not an open Telnet service in general. Nevertheless, the function is called anyway, as there is no downside to calling it. The script then uses the *checkIfTelnetWorks()* function described above to check if the current Telnet service is running on the machine on port 23 or not. If not, the script returns from the *listenToDeployer()* function and logs accordingly, as no measurements are required. Otherwise, the port change is initiated by another function called *changePort()*, which is described in more detail above. This function changes the port of the Telnet service from 23 to another random port sent by the *Deployer Server*, checks if this port change worked, and also changes the Telnet service back to port 23 after a given number of seconds. After this function finished, the socket is closed and the script returns from the *listenToDeployer()* function. The code block for changing the port of the Telnet service on the clients can be seen in Algorithm 4.10.

Algorithm 4.10 The Code Block for Changing the Port of the Telnet Service of a Client From the *listenToDeployer()* Function in Pseudocode. Everything Related to the IP Address Change has Been Ommited Here.

```
1  def listenToDeployer(HOST, PORT ):
2    CREATE listening socket
3    WHILE True:
4      SET count to 0
5      RECEIVE max attempt for trying to find port
6      SEND unimportant answer
7      WHILE True:
8        RECEIVE command from server
9        SPLIT command to movingParameter and movingParameterValue
10       IF movingparamter is "IP":
11         // NOTE: omitted the code for the IP address change
12       ELSE IF movingParameter is port and count <= max attempt:
13         INCREMENT count
14         CHECK if movingParameterValue (port) is used
15         IF port is used:
16           SEND that port is already used
17           CONTINUE with WHILE loop
18         ELSE
19           CALL function to get telnet port
20           IF not possible:
21             SEND error
22             CLOSE socket
23             RETURN
24           END IF
```

```
25            SEND that port is unused
26            CALL function to kill existing telnet processes
27            CALL function to check if telnet listens on port 23
28            SEND message to get the time to change back
29            RECEIVE time to change back
30            CALL function to change port
31            CLOSE socket
32            RETURN
33          ELSE
34            SEND max attempt are exceeded
35            CLOSE socket
36            RETURN
37          ENDIF
```

Chapter 5

Evaluation

This chapter presents the evaluation of the implemented Moving Target Defense (MTD) framework with its techniques based on different criteria. First, the methodology of the evaluation is explained, followed by the missing prerequisites for the evaluation to work. The chapter then presents the evaluation process and its results. These results are discussed in Chapter 6.

5.1 Evaluation Methodology

The evaluation of the solution was quite complex because the whole setup had so many parts/scripts distributed across three different VMs. Ultimately, three different metrics were to be evaluated. The first and main evaluation metric is the total number of seconds that the two machines in the network are infected. This is reasonable as the goal of this thesis is to mitigate/prevent the infection of IoT devices and to show that a collaborative approach yields better results than a non-collaborative one. Thus, the fewer seconds the malware is running on the devices, the better. The other two metrics are defender metrics mentioned by [60], who investigated MTD research trends. Both metrics fall into the system performance category.

The first system performance metric is the amount of time that incoming and outgoing communication to and from the device is interrupted. This can be further divided into the interruption caused by the IP address change and the interruption caused by the Telnet service port change. For the IP address change, only the outgoing connection was tested. The reason for this is that the incoming connection would only work if there was a mechanism to keep track of which machine has which IP address, similar to the one suggested by [37]. However, outgoing connections are more critical, as IoT devices often need to pass on collected data. Regarding the Telnet service port change, incoming and outgoing connections to and from the Telnet port were evaluated. The other system performance metric is the CPU/RAM usage of the deployed MTD techniques, which is essential due to the hardware limitations of IoT devices.

Other potentially important metrics [60] such as Quality of Service (QoS) to users or strategy switching costs were not included. This exclusion is due to the fact that these metrics were either out of scope or not feasible (e.g. power consumption). The three measured metrics were therefore:

1. Total number of seconds that the machines are infected
2. Interruption of network availability of the machines
3. CPU/RAM usage of the framework and the executing MTD techniques

These metrics were measured in three different environments/systems for comparison. All environments used the VM structure already described, but differed in the solution applied and their deployment strategy (i.e. proactive vs. reactive deployment of MTD techniques). Environment 1 used a reactive, modified solution that did not include a cooperative component. This was done by only enabling the IP address change of infected machines. This approach already exists [35], but the implementation code of the approaches are independent. This environment allowed the collection of data from a solution that uses a non-cooperative and reactive defence approach, and served as a baseline for this evaluation.

In contrast, Environment 2 used the IP address and the port change to reactively mitigate/prevent Bashlite. This environment allowed to collect data from a solution that uses a cooperative and reactive approach. Environment 3 used the cooperative approach (IP address change and Telnet service port change), but executed this proactively rather than reactively. This provided an interesting insight into the trade-offs/differences between the proactive and reactive approaches. While the reactive approaches were executed 10 seconds after Bashlite was executed, the proactive MTD techniques were executed every 60 seconds, regardless of whether Bashlite was active or not. The 60 seconds were initially chosen randomly, but proved to be a reasonable value given the results. Furthermore, the goal was to show the typical differences between the reactive and proactive approaches, rather than to determine an optimal time to move, as this would be influenced by many more factors of the system. Table 5.1 summarises the three environments.

The data was collected over 30 runs (entire infection/mitigation process) in each environment and is presented in Section 5.3. The first five of these 30 runs were removed from the data as they were used to warm up the entire system. Section 5.2.1 describes the data collection in more detail.

5.2 Prerequisites for the Evaluation

Even though the implemented MTD mechanism worked, its code had to be adapted on several occasions. This involved three main tasks:

1. Writing scripts to measure and log the metrics, such as how long Bashlite has been running on the machine.

	Proactive vs. Reactive	Cooperative vs. non-Cooperative	Applied MTD Techniques
Environment 1	Reactive (10 seconds after Bashlite execution)	Non-cooperative	IP address change of infected
Environment 2	Reactive (10 seconds after Bashlite execution)	Cooperative	IP address change of infected and Telnet Service Port change of susceptible
Environment 3	Proactive (Every 60 seconds)	Cooperative	IP address change of infected and Telnet Service Port change of susceptible

Table 5.1: A Table Showing the Characteristics of Three Different Environments in Which the Previously Defined Metrics are Measured.

2. Automating the code as much as possible. To get a reasonable amount of data, the infection/mitigation process had to be run multiple times, which was not feasible by hand. The idea was to be able to run the whole process once and then repeat it n times.

3. Modifying the current scripts to fit the different environments. This included, for example, the adaptation that no Telnet service port change was triggered in Environment 1.

The following subsections briefly describe how these tasks were accomplished. However, first an important aside regarding the Bashlite client. There was the problem that it was not possible to check whether a Bashlite client was still connected to the Bashlite server or not. To solve this, the killing of the Bashlite client in the IP address change technique remained in the code. This was justified by extensive testing and the existing literature [35] regarding the interruption of the connection between the server and the client, and because the evaluation would simply not have been possible otherwise. Moreover, Bashlite is killed as soon as the client loses connection to the server, so there is only a negligible difference in time. This allowed the simplification that if a Bashlite client was running on the system, it was also connected to the server and vice versa.

5.2.1 Data Collection

Having solved this problem, a mixture of Bash and Python scripts were written to log the data into some .csv files, which were later analysed and plotted using Python. The reason

for using both types of script was that sometimes one or the other was more suitable. This resulted in several files.

The *stopTimeOfClient.sh* script checks every 0.05 seconds whether the client process is running on the machine. If it is, the script stores the time of the start of the client, waits for it to finish and writes all the information to a *bashliteAnalysis.csv* file. This script is continuous and also indicates which Bashlite run the system is on.

The *CPURamAnalysis.py* script uses *psutil* to monitor the CPU and RAM usage of the system and writes this to a CPURamAnalysis.csv file. The original idea was to specifically measure the CPU and RAM usage of the script that executes the MTD techniques on each machine. However, as the CPU usage of the script was almost always 0% (measured with Bash/top and Python/psutil to exclude errors), a graph showing this would not give much insight. Thus, only the RAM usage was measured with *psutil* since this was giving correct and measurable results. The CPU usage of the infected and susceptible machines was measured once with the MTD mechanism deployed and techniques executing, and once without the MTD mechanism deployed. This allowed to compare the case with MTD to the case without MTD on the respective machine. Both measurements were run for 15 infection/mitigation runs, equivalent to 30 minutes.

The *sendPacketsToServer.sh* script monitors outgoing packet losses. This is achieved by sending simple one-packet pings to the server machine every 0.5 seconds. The script also keeps track of the current Bashlite run. This gives interesting insights for the different environments. In Environment 1 and 2, outgoing packet losses should only occur once per Bashlite run, as the MTD techniques are executed reactively. In Environment 3, the MTD techniques are executed proactively, and therefore multiple packet losses are possible during a Bashlite run. All this information is written to a *packetLoss.csv* file.

The *telnetToLocal.py* and *telnetToLocalOnlyIP.sh* scripts were written to track the interruption of the incoming Telnet service from the susceptible machine. The former was created first and uses *sockets* (or *telnetlib*) to check whether port 23 of the executing machine is listening or not. This produced reliable results, but there was an issue in the non-cooperative environment due to the two possible IP addresses the susceptible machine could have. Sockets in general were too slow to either get the host IP address or try to connect to both possible IP addresses, so a shell script that uses *netcat* to check whether port 23 of both possible IP addresses are listening was more suitable. All this information is logged to a file called *telnetToLocal.csv* or *telnetToLocalOnlyIP.csv* depending on which script was used.

The *telnetToServer.py* file is the last script and monitors whether outgoing telnet connections are possible or not. It does this using the *telnetlib* module as in this case, it was essential to use Telnet as the communication protocol for the outgoing connection.

5.2.2 Automation

Regarding automation, the implementation code already fulfilled most of the requirements. Only four issues remained. The first was that the script *sendToDeployerClient.py*

from VM1 needed some adaptations to separate the start of Bashlite from the start of the defence technique (notifying the *Deployer Client*). One reason for this was the parent/child relationship between Bashlite and the Python script, which made it impossible to kill Bashlite (the child) while the Python script (the parent) was still running. Creating another Bash script called *startBashAndSendToDepClient.sh* solved this problem. This file contains a while loop with each iteration being an entire infection/mitigation process. First, the while loop starts the Bashlite client and then sleeps for 10 seconds, mimicking the time it takes to detect Bashlite. After this time it starts the *sendToDeployerClient.py* in the background and then sleeps again for 110 seconds. This time period could also be shorter, but it ensured that the previous infection process is completed with all the corresponding MTD techniques.

The second problem was that the *sendToDeployerClient.py* script had to be triggered somehow on VM2 as well. It was not possible to use the same script as for VM1 (*startBashAndSendToDepClient.sh*), since this started Bashlite manually on VM1 and VM2 needed to detect it somehow. The solution was a Bash script that runs continuously and checks if Bashlite is running. If Bashlite is running for more than 10 seconds, the script executes *sendToDeployerClient.py* which then initialises the MTD mechanism. This functionality was implemented with a shell script called *checkForBashliteAndStartMTD.sh*.

The third encountered issue was that the Bashlite server would only allow one infection, after which a restart of the server was required. This was added at the beginning of the thesis to improve stability, as the Bashlite client was constantly sending the same report about the susceptible machine, causing the Bashlite server to crash. Since it was not an option to always restart the server manually, the Bashlite client was adapted to send the report only once, and thus the restriction on the Bashlite server could be removed.

The fourth issue was that about 15 scripts per environment would have had to be started to get everything running. To avoid this, a Bash script (*runAll.sh*) was written containing all the startup instructions for each machine and environment. This resulted in three different startup scripts on three different machines, which was acceptable. Additionally, the counterparts for terminating all scripts started by *runAll.sh* were created. This allowed to quickly start all the scripts, let them collect data and then terminate them effortlessly.

5.2.3 Creating Evaluation Environments

Finally, the different environments/systems had to be created. The first environment was the non-cooperative one, which only involved changing the IP addresses of the infected machines. The only required server-side adaptation was to instruct the *Deployer Server* not to initiate any Telnet service port changes. This was done by replacing the array containing all the IP addresses that were potentially going to receive a port change with an empty array.

On the client side, there was the problem that the Bashlite client was only scanning one IP address for a susceptible machine, so as soon as the IP address change was executed on VM2, the Bashlite client on VM1 would no longer find VM2 in the next Bashlite run. To solve this, two hardcoded IP addresses replaced the IP address variable received from

the *Deployer Server*. Thus, VM2 simply switches between these two IP addresses instead of migrating to the one sent by the *Deployer Server*. This allowed adapting the Bashlite scanner of the client of VM1 to try only two different IP addresses, which worked perfectly. Note that this was only needed for the evaluation as the Bashlite implementation is limited. If it were not for the evaluation, or if Bashlite was scanning in a more sophisticated way, this adaptation would not be required at all. Besides, this did not make any difference to the data collected.

The second environment was the original idea, which was Bashlite together with a cooperative MTD approach. Since this usecase was the goal from the start, nothing had to be changed. The third environment involved Bashlite with the proactive MTD execution. To create this environment, the only required adaptation was to split the *startBashAndSendToDepClient.sh* script into two subscripts. The first script starts the MTD mechanism every 60 seconds, the second script starts Bashlite randomly at an interval between 60 and 90 seconds. These two scripts are then executed by the *runAll.sh* script.

5.3 Results

This section presents the results of the evaluation. It is divided into the three metrics mentioned in Section 5.1, namely the duration of infection activity in the network, the interruption of availability, and the CPU/RAM usage.

5.3.1 Duration of Infection Activity in the Network

The duration of infection activity was the most important metric as it defined whether the defence mechanism was working or not. Figure 5.1 shows the results for Environment 1, which was the non-cooperative and reactive solution. On the x-axis is the current run (infection/mitigation process). The blue bar of the stacked bars shows the number of seconds that VM1 was infected with Bashlite, meaning that Bashlite lasted approximately 29 seconds on VM1 in each run. The orange bar shows the number of seconds that VM2 was infected, which was also approximately 29 seconds per run. This gave an average of 57.6 seconds of total infection activity on the two VMs per run. It is evident that in this non-cooperative approach, the spreading to the susceptible machine could not be prevented at all. Bashlite was still mitigated, but each machine had to clean itself after the infection had occurred.

Figure 5.2 shows Environment 2, which was the cooperative and reactive environment. Again, the blue bars represent the number of seconds that VM1 was infected, the orange bars would represent the time that VM2 would have been infected if this had occurred. It is evident that in the cooperative and reactive environment the spreading to VM2 was prevented in every run. This resulted in an average of about 28.8 seconds of total infection activity on the two VMs, which is approximately half of the time of the uncooperative and reactive environment.

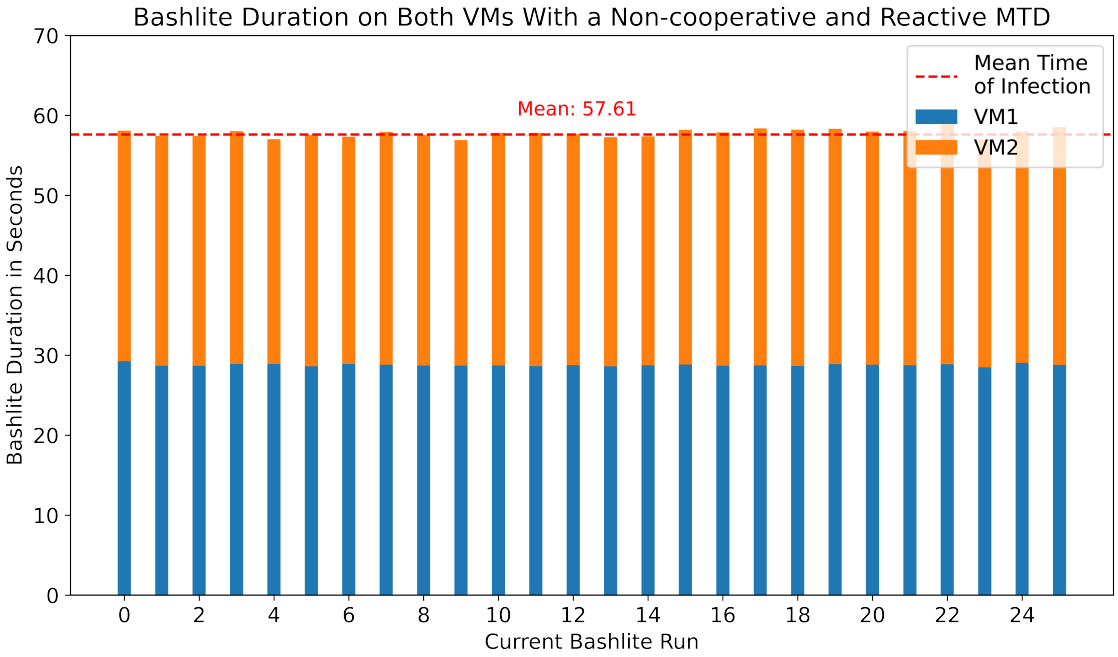


Figure 5.1: The Duration of Infection Activity on Both Virtual Machines in the Non-cooperative and Reactive Environment.

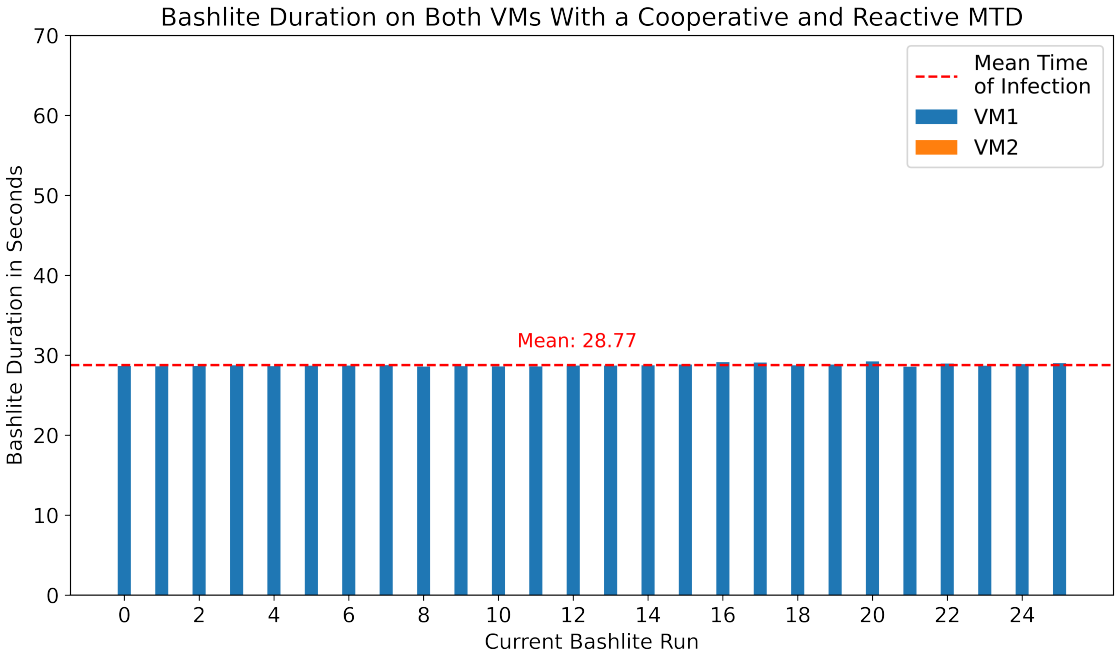


Figure 5.2: The Duration of Infection Activity on Both Virtual Machines in the Cooperative and Reactive Environment.

Figure 5.3 shows Environment 3, which was the cooperative but proactive environment. Unlike the previous two environments, the Bashlite duration was different for each run. The mean duration was slightly higher than the mean duration of the cooperative and reactive environment, but much lower than that of the non-cooperative and reactive environment.

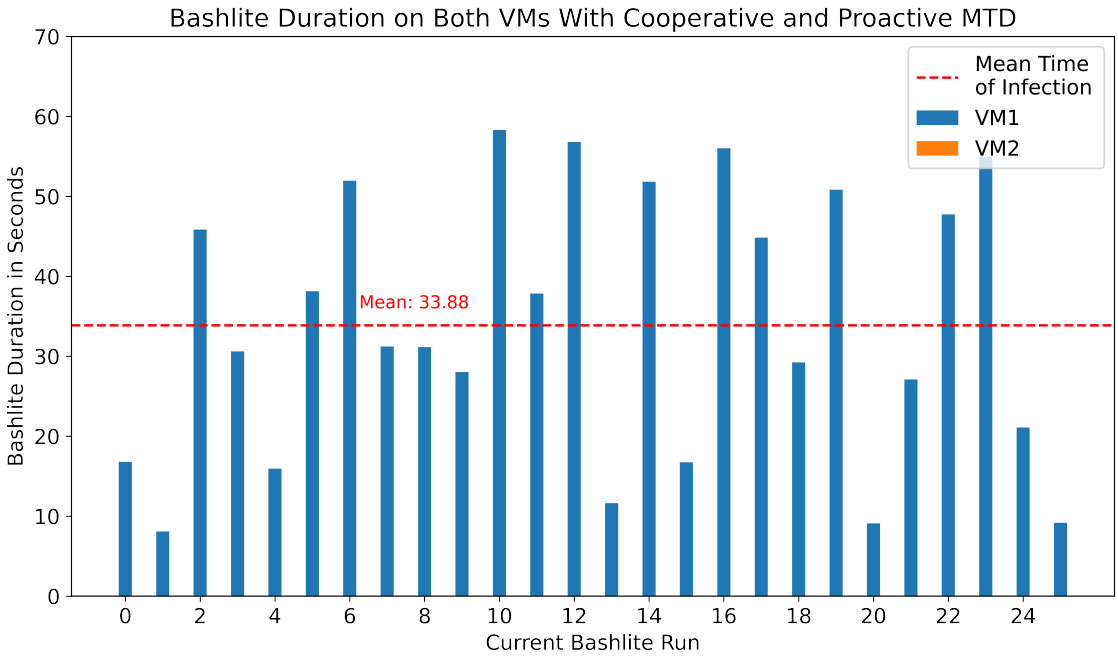


Figure 5.3: The Duration of Infection Activity on Both Virtual Machines in the Cooperative and Proactive Environment.

Figure 5.4 shows the duration of the average total infection activity on VM1 and VM2 over 25 runs. The cooperative and reactive environment had the shortest duration of infection activity, the cooperative and proactive environment followed immediately and the non-cooperative and reactive environment had by far the longest average duration of infection activity. This was due to the infection of the susceptible machine.

5.3.2 Interruption of Availability

Figure 5.5 shows the share of packet losses out of the total packets sent by the two VMs in all three environments. Environment 1, which executed the IP address change on both machines on every run due to the uncooperative nature, had the highest packet loss with about 4.2%, the remaining 95.8% were successfully delivered. Environment 2, which consisted of the cooperative and reactive MTD, had by far the lowest packet loss with about 2.1%. Environment 3 with the cooperative and proactive approach had a similar packet loss as Environment 1.

The incoming Telnet connection showed a different picture than the packet losses, as shown in Figure 5.6. Environment 1 had by far the lowest share (4.2%) of failed incoming

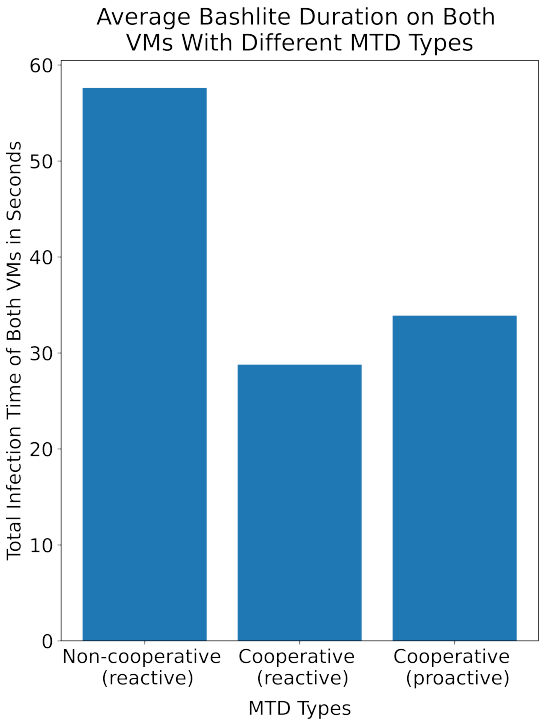


Figure 5.4: A Summary of all Three Environments and Their Respective Infection Duration.

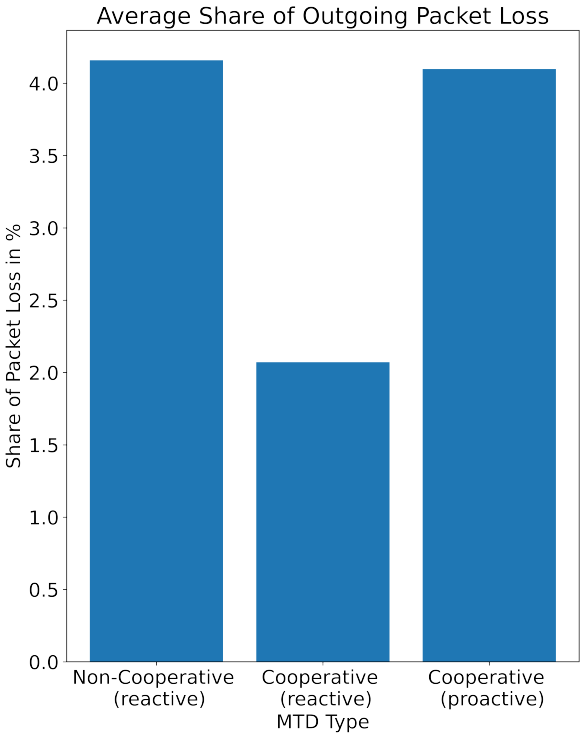


Figure 5.5: A Summary of all Three Environments and Their Respective Average Share of Outgoing Packet Losses From all Packets Sent.

Telnet connections of all incoming Telnet connections. This was due to the fact that no Telnet port changes were involved here (only the IP address change). Environment 2 was in the middle with a share of 31.6% failed incoming Telnet connections and Environment 3 had by far the highest share of failed Telnet connections with 61.4%.

In addition to the incoming Telnet connections, the outgoing Telnet interruptions were also measured. It was evident that changing the Telnet port did not interrupt or prevent any outgoing connections. The only time that outgoing Telnet connections were not possible was when the IP address of the device changed. Therefore, the interruption of outgoing Telnet connections was the same as the share of outgoing packet losses shown in Figure 5.5 for every environment. This means that in the non-cooperative and reactive environment and the cooperative but proactive environment, approximately 4.2% of outgoing Telnet connections failed. In the cooperative and reactive environment, around 2.1% of outgoing Telnet connection failed.

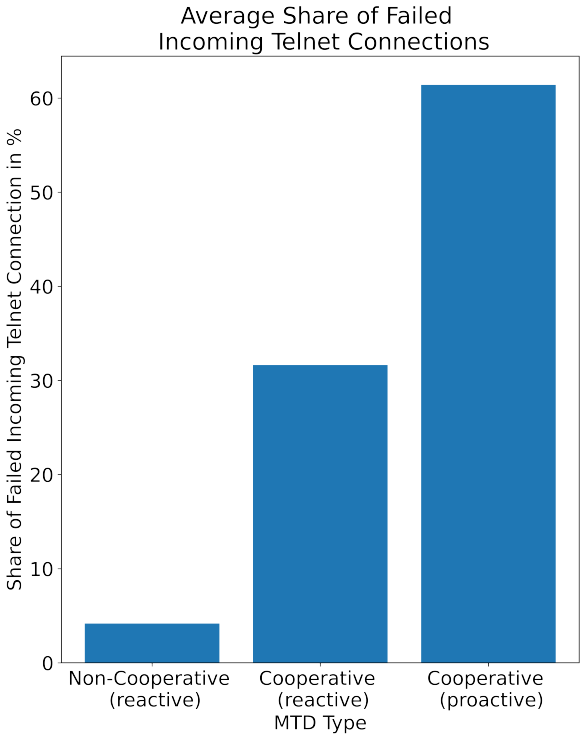


Figure 5.6: A Summary of all Three Environments and Their Respective Average Share of Failed Incoming Telnet Connections.

5.3.3 CPU and RAM Usage

Figure 5.7 shows the constant RAM usage of the MTD mechanism over 30 minutes with the MTD techniques executed multiple times on both VMs. VM2 used a minimal amount more RAM, but this difference is negligible as it is in the byte range and the machines have gigabytes of available RAM. So in general, the mechanism's RAM usage was minimal.

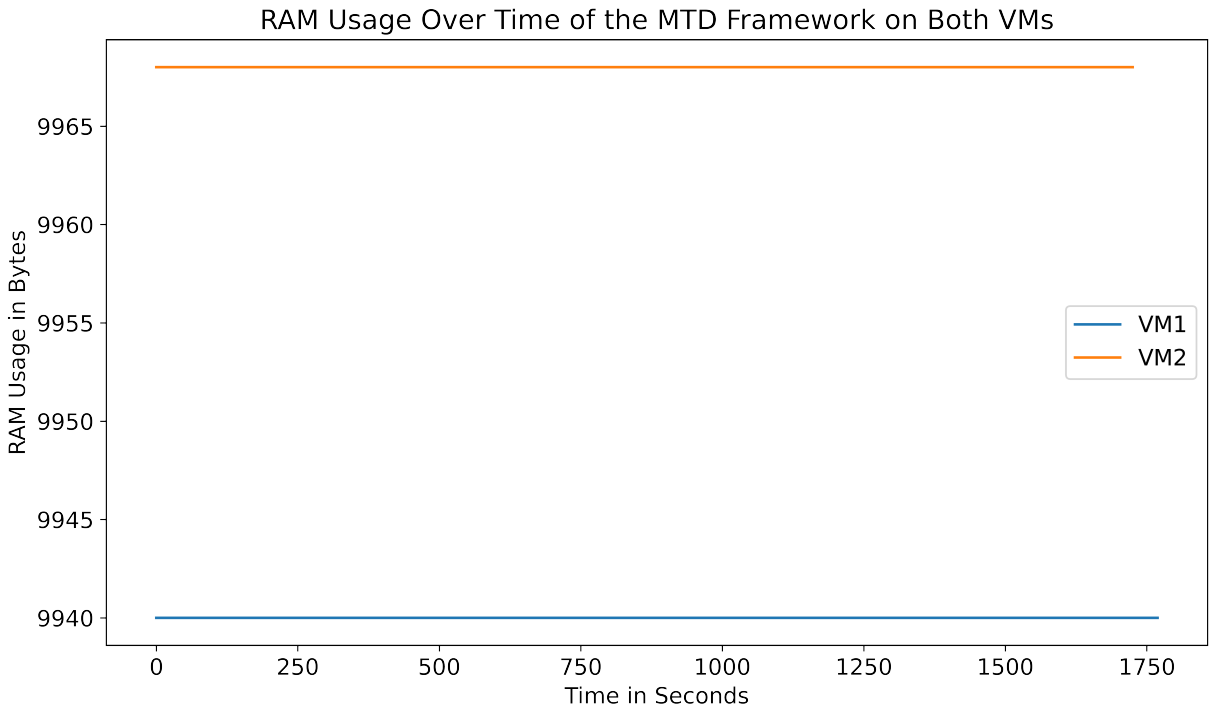
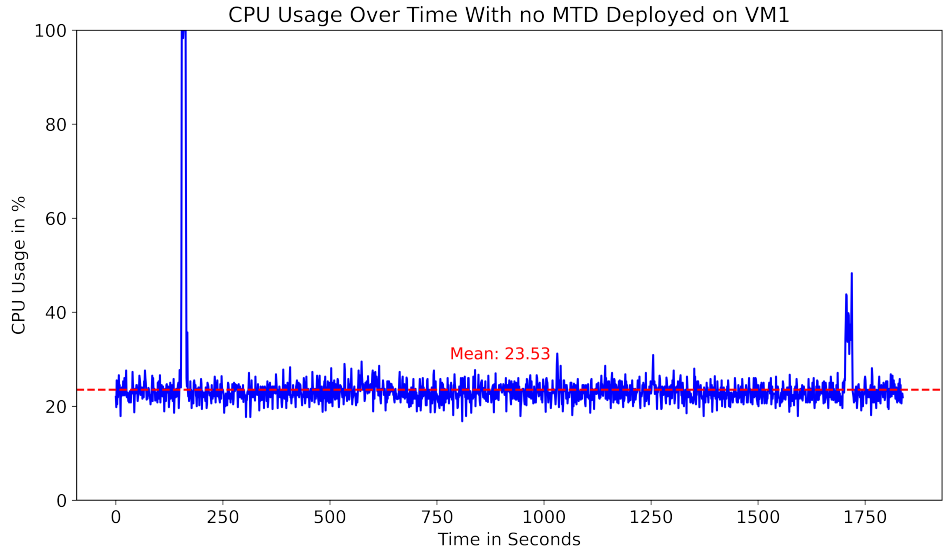


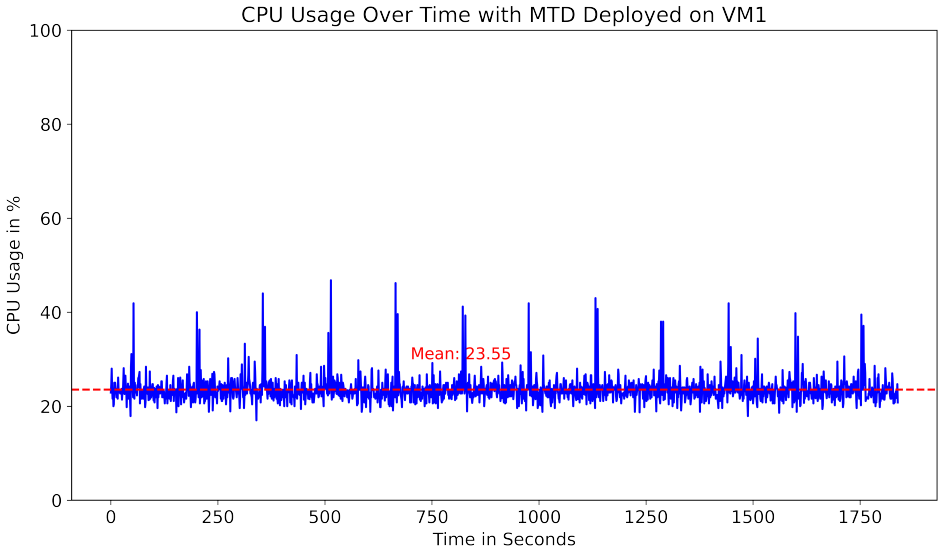
Figure 5.7: The RAM Usage of the Running MTD Framework with Multiple Executions of the MTD Techniques on Both VMs.

Figure 5.8a shows the CPU usage of VM1 without the MTD deployed. The CPU usage varied between 20% and 32% with only a few outliers that are higher. In this case, no other program or script was started except the measurement script, so these outliers must be caused by the operating system or some other automatically started process. The average CPU usage was 23.53%. Figure 5.8b shows the same information, but this time with the MTD framework deployed and the MTD techniques executing multiple times. On VM1 the MTD technique performed was the IP address change. It is clear that the fluctuation was higher with the MTD framework deployed, ranging from 20% to about 40%. The high peaks were periodic, but also rather short. The average CPU usage was 0.02% higher with the MTD mechanism deployed than without it.

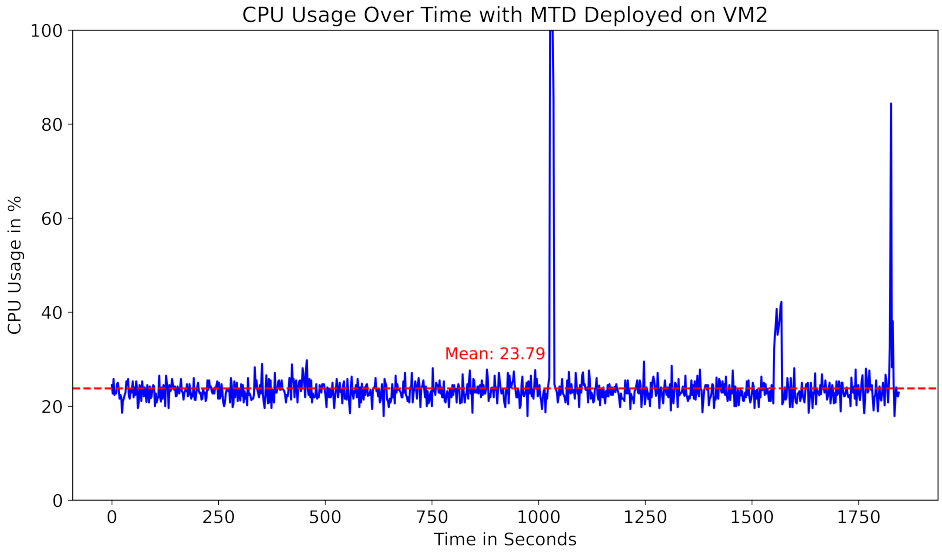
Figure 5.9a and Figure 5.9b give the same information, but for VM2, which performed the Telnet service port change. Again, there was a range of 20% to 30% with a few higher outliers in the base case. The average was 23.79%, which was slightly higher than the average CPU usage of the base case of VM1. The CPU usage of VM2 with the MTD framework deployed and the techniques executing also showed more fluctuation than the base case of VM2. The average CPU usage of the deployed case was also slightly higher, namely 0.43%. Although there were fewer high peaks on VM2 with the MTD mechanism deployed than in the respective case of VM1, the average CPU usage was higher on VM2 than on VM1.



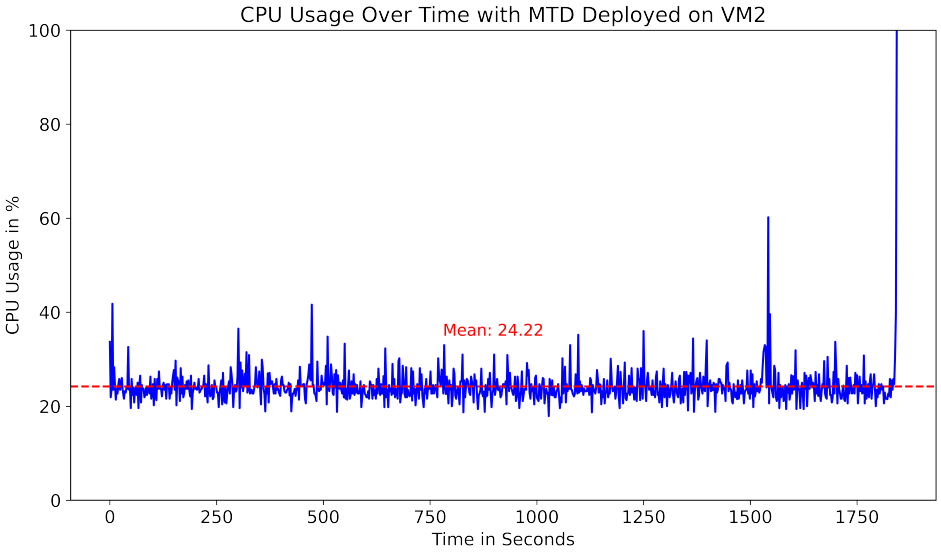
(a) The CPU Usage of VM1 Without the MTD Mechanism Deployed.



(b) The CPU Usage of VM1 With the MTD Mechanism Deployed. This Machine Executed the IP Address Change.



(a) The CPU Usage of VM2 Without the MTD Mechanism Deployed.



(b) The CPU Usage of VM2 With the MTD Mechanism Deployed. This Machine Executed the Telnet Service Port Change.

Chapter 6

Discussion

This chapter presents the discussion of the results shown in Section 5.3. Additionally, some of the limitations of the thesis are outlined.

6.1 Interpretation of Results

The results of the evaluation turned out as expected. The cooperative and reactive approach provided the best defence against Bashlite, as the overall infection activity was the shortest. It also made sense that the variance of the Bashlite durations was so low in the reactive approaches, because there is no stochastic process involved in the defence mechanisms. In contrast, the proactive environment involves randomness, resulting in different overall infection times for each run. Thus, as long as the MTD techniques execution is reactive and the hardware does not reach its limit, which is unlikely, the mitigation process will have a similar mitigation time for each run.

This mitigation time was around 29 seconds in the evaluation, which was surprisingly high. 10 seconds are needed for detection [35], the remaining 19 seconds are needed for the mitigation process. A large part of these 19 seconds was caused by the *nmap* scan on the *MTD Deployer Server*. Minimising this scan time was beyond the scope of this thesis, but it would be achievable [61] and should be implemented in a more sophisticated solution. Another convenient way to reduce the *nmap* search time would be to change the range of possible IP addresses in the *MTD Deployer Server* configuration file. The evaluation was done with a range of 90 IP addresses, whether this is a reasonable number depends on the system in which the MTD would be deployed.

The detection time is another crucial variable. In this thesis, a detection time of around 10 seconds was used, as it was shown that Bashlite could be detected in this time [35]. If this time were longer (e.g. with another malware), it is possible that a reactive approach would not be able to prevent the malware from spreading. This could force a user to use a proactive defence mechanism instead of a reactive one. The proactive approach used in the evaluation was successful in preventing Bashlite from spreading to the susceptible machine with the variables selected. This is by no means certain. Several variables are

decisive whether the spreading of Bashlite can be prevented, the most important ones are: the time interval after which Bashlite is executed randomly, the execution interval of the MTD mechanism, and the time after which the Telnet service port is moved back to port 23.

If the system protection was the only criterion, it would be best to run the MTD techniques at a very short interval. For example, an MTD mechanism (e.g. IP address change) executed every second would result in an extremely short overall infection activity. While this is desirable, it is not feasible as the connections to and from the device would always be interrupted, defeating the purpose of the device (e.g. a sensor collecting data). Thus, there is usually a trade-off between system availability and security.

This trade-off between availability and security was also reflected in the results. In terms of outgoing packet losses, the non-cooperative and reactive approach had the highest share of packet loss out of the total number of packets sent. This is evident as both machines had to change their IP addresses, which is the cause of the packet loss.

The share of failed packet losses can also be approximated mathematically. In this non-cooperative and reactive environment, a run took 120 seconds on each machine, resulting in a total of 240 seconds in the system. Restarting the Ethernet adapter to force the machine to use the new IP address takes about 5 seconds per machine, making a total of 10 seconds. 10 seconds is about 4.2% of 240, which is the approximate interruption percentage. This also explains why the packet loss share of all packets sent in the cooperative and reactive environment was exactly half of the packet loss in the non-cooperative and reactive environment. Again, the total time was 240 seconds, but this time only one machine changed its IP address. 5 seconds out of 240 seconds is 2.1%. The same calculations can be done for the cooperative but proactive environment. Here only one machine had to change its IP address, but it did so every 60 seconds due to the proactive approach. As there are two machines again, this gives a total of 120 seconds in the system. 5 seconds is about 4.2% of 120 seconds.

So far, the cooperative and reactive approach performed best. This approach had the lowest total infection time and the lowest share of outgoing packet losses. This was different when looking at the share of failed incoming Telnet connections. Here the non-cooperative and reactive approach performed best. This is evident as this approach did not initiate port changes, which are the main cause of failed incoming Telnet connections. Although not clearly visible in the figures, the share of failed incoming Telnet connections for this non-cooperative and reactive environment is identical to the share of its packet loss (4.2%). This is due to the reboot of the Ethernet adapter, as a working IP address is a prerequisite for successful Telnet connections.

The cooperative and reactive environment had a much higher interruption share of incoming Telnet connections due to the moved Telnet service port. It is important to note that this share was heavily influenced by the defined time to switch back to port 23, which can be specified in the *MTD Deployer Server* configuration file. The 30 seconds selected for the evaluation were chosen because Bashlite would definitely be rendered harmless on the infected device and the port could therefore be moved back after 30 seconds. Again, the share of failed incoming Telnet connections was twice as high in the cooperative but

proactive environment as in the cooperative and reactive environment. This is because the former ran the MTD every 60 seconds and the latter every 120 seconds.

As mentioned earlier, there is a trade-off between security and availability and all the deployment strategies had their advantages. However, the only advantage that the non-cooperative reactive system had over the cooperative reactive system is the lower rate of failed incoming Telnet connections. Although it is unlikely that the availability of a device's Telnet port is more important than security and the outgoing packet loss rate, a solution in this case would be to use an uncooperative defence.

The cooperative and reactive approach also showed better results in the evaluation than the cooperative but proactive approach. However, this case is more complex as it depends on various factors such as the detection time of the malware or again the trade-off between security and availability. The proactive approach could theoretically provide better security results than the reactive approach, but this would also increase the interruption of the services. In a real system, the requirements of the underlying system would have to be compared with the advantages and disadvantages of the MTD solution. One way of doing this is to calculate the interruption time and decide which approach is more suitable for the system at hand.

In general, the cooperative and reactive approach can be recommended for most systems, as this will only run the MTD techniques if Bashlite has been found on the system. This avoids unnecessary interruption of the system that would occur with the proactive approach. However, it is impossible to make a final statement, as the choice also depends on the malware. If a really aggressive malware is trying to infect and possibly destroy the devices, a proactive approach may still be more reasonable. Nevertheless, the possibilities offered by a cooperative defence mechanism are very promising.

Even in case a proactive approach is the required solution, there is no need to worry about the resources of the device, as it was found that the MTD framework and executed techniques used only a minimal amount of hardware resources. This is evident from the fact that it does not take many hardware resources to execute the *sed* command to replace something in a file and then restart either the Ethernet adapter or the *inetutils-inetd* service. The peaks seen in the CPU usage of VM1 with the MTD deployed are almost certainly caused by the restart of VM1's Ethernet adapter. It is not clear what caused the peaks of VM2. The reason may be the restart of the *inetutils-inetd* service, but as they were not as periodic as VM1's peaks, it is difficult to determine. However, these peaks were extremely short and therefore not a problem even for resource-constrained devices. In terms of RAM usage, the results were similar. The MTD framework and the executed techniques used only about 10,000 bytes of RAM. This is negligible compared to the 3.7 GB the machines had at their disposal, as it is less than 0.0003%.

The results of this evaluation could also be usefully combined with two of the frameworks presented in Section 3.2. The first was the framework that helps to answer the design questions (What, How, When) for an MTD mechanism. The What and the How are already determined by the implemented MTD mechanism, as well as the When for the reactive case, since this is determined by the detection time. However, the When for the proactive approach could definitely be determined with the help of this framework. The second framework was the IANVS framework, which aims to help implement MTD

techniques in distributed systems. Whether this is needed or not depends on the system. As long as the connections of the IoT devices are outgoing and the target is static, such a mechanism is not needed because the IoT device can still connect to the target regardless of what IP address the IoT device currently has. However, if a device needs to connect to the IoT device, such a mechanism needs to be implemented.

6.2 Limitations

There are some important limitations in several aspects of this thesis. The first is the operating system of the virtual machines. As described, the initial setup was with the Ubuntu operating system, which had to be changed to the Raspberry OS in order to run Bashlite. This change caused the original implementation of the IP address change to fail. Although the Python parts of the solution should work independently of the OS, it is possible that the Bash commands executed in the Python code may need to be adapted on a different OS. This is especially important as there are many different potential operating systems for constrained devices, as shown in Section 2.2.3.

Another limitation is the chosen malware. In this thesis, Bashlite was used as the malware to work with. However, there exist more sophisticated malware that may be able to evade the defence mechanisms presented. Although this could only be evaluated through the code on GitHub, a malware such as Mirai has implemented the functionality to reconnect to its server when the client does not receive a response from the server. This makes the IP address change more of a short-term obstacle than a long-term solution. Additionally, it is not entirely clear how the IP address change would perform against a P2P malware such as the HEH malware described in Section 2.3.2. At the very least, the Telnet port change technique is, in theory, a well working defence against malware such as HEH or Mirai. However, this also needs to be thoroughly tested in practice. Additionally, it is worth noting that malware can also target a device's SSH port. This thesis has only focused on the Telnet port.

As already described in more detail in the previous Chapter (6), the results of the evaluation were strongly influenced by the chosen variables and the configuration file (e.g. the proactive execution rhythm or the IP address range to be scanned by *nmap*). Although this is normal in such an evaluation, it is important to emphasise this when talking about the limitations. Even though the variables selected for this thesis were carefully chosen, it is not possible to draw an absolute conclusion from the result. It is clear that in most cases it makes sense to use a cooperative and reactive approach where possible, but there may be environments/systems where the cooperative and proactive approach is more suitable. Various variables determine how well an MTD approach performs, and different MTD approaches may have different advantages and disadvantages. It is therefore important to tailor the chosen MTD approach to the requirements of the system that is to be protected by the MTD solution.

Chapter 7

Conclusion

This thesis proposed and implemented a cooperative Moving Target Defence (MTD) framework for IoT devices along with two MTD techniques. IoT devices are inherently susceptible to malware for several reasons, including poor maintenance and neglect of security [4]. This insecurity, combined with the immense number of existing IoT devices [2], makes these devices a popular target for malware. Infected IoT devices can be used for various malicious behaviours, the most common use case being botnets to perform DDoS attacks, for example. One way to defend IoT devices against malware is MTD, that attempts to change the attack surface of a system to defend it against attackers [7]. [35] showed that it is possible to disconnect a Bashlite bot from the Bashlite server by changing the private IP address of the client. The solution proposed in this thesis builds on this by adding a cooperative component, as this provides a significant advantage to the defence mechanism.

In order to find an effective defence, a suitable malware had to be found. The choice fell on Bashlite, a fairly well-known IoT malware. Even though Bashlite was available on code repositories, some adaptations had to be made to the code to provide, among other things, the spreading functionality. With Bashlite working, the cooperative MTD mechanism could be developed. As a test bed, three virtual machines running Raspberry OS were used, one as a server machine and the other two to mimic the IoT devices. Of these two mimicking machines, one was manually infected (VM1) and the other was the susceptible machine (VM2) that was eventually infected through the spreading of Bashlite.

The implemented solution consists of two different MTD techniques, which are the IP address change and the Telnet service port change, as IoT malware often infects devices via Telnet. The currently infected device performs an IP address change to disconnect itself from the Bashlite server, making communication with the server impossible. All other devices in a given IP address range should change their Telnet service port for a specified time to hide from Bashlite.

After the implementation, the solution was evaluated using several metrics, including the overall infection time of Bashlite on the two virtual machines, the network interruption caused by the execution of the MTD, and the RAM and CPU usage. These three metrics were measured in three different environments for comparison. The first environment was

an non-cooperative and reactive MTD approach. This environment executed the MTD techniques 10 seconds after Bashlite was detected, but did not include a cooperative component. This means that Bashlite first had to be detected on the machine in order to disrupt its connection to the Bashlite server. The second environment was a cooperative and reactive MTD approach. Again, the MTD techniques were executed 10 seconds after Bashlite was detected, but here with the cooperative component. This was the solution developed in this thesis. The third and final environment was a cooperative but proactive environment. This was used for comparison and to show the trade-off between a proactive and a reactive MTD approach. This proactive approach ran the MTD techniques after a specified time interval (1 minute), regardless of whether Bashlite was on the system or not.

The results showed that in the non-cooperative and reactive approach, the susceptible machine was infected on every run of Bashlite. This meant that both machines had to change their IP address to disconnect from the Bashlite server, resulting in each machine being infected for approximately 29 seconds. The cooperative and reactive approach completely prevented the spreading of Bashlite to the susceptible machine by changing the Telnet service port of the susceptible machine. Therefore, only the manually infected machine was infected for 29 seconds and the susceptible machine was not infected at all. This meant that the overall duration of infection activity on the two VMs was halved. Although not tested, this would be even more significant with more susceptible machines, as any infection other than the first could be prevented. Finally, the cooperative but proactive approach also prevented Bashlite from spreading to the vulnerable machine with the chosen configuration. However, it also showed a large variance in Bashlite duration on the first machine, as randomness partly determines after what time the MTD techniques are initiated. This resulted in an average Bashlite duration of 34 seconds on VM1 and 0 seconds on VM2. The results of this approach are strongly influenced by the configuration chosen, i.e. the interval at which the MTD techniques are initiated and other variables. It is also possible, for example, that the spreading of Bashlite cannot be prevented at all if the interval between the execution of the MTD techniques is too long.

Besides the overall infection time, the downtime of the machines is an important metric. The results clearly showed that a cooperative and reactive approach gave better results in terms of downtime than the uncooperative and reactive approach, as long as Telnet is not required on the susceptible machine. The cooperative and reactive approach also caused much less downtime (about 50%) than the proactive and cooperative approach, which was to be expected. The final metric was the CPU and RAM usage of the MTD techniques. It turned out that both machines used on average a maximum of 0.43% more CPU with the MTD mechanism deployed than without it and that the RAM usage of 10000 MB on each machine was negligible.

Although the cooperative and reactive approach performed best in the evaluation, the other approaches should also be kept in mind. This is due to the different variables that should be considered when deciding on the most suitable MTD approach. Each approach has different advantages and disadvantages. For example, a short-interval proactive approach may be the best solution if the overall infection time of the system is to be minimized, and high downtime of connections to and from the machines is not an issue. In addition, it may be that the malware in question is not (yet) detectable, in which case the

proactive approach would be required anyway. However, for most systems, the cooperative and reactive approach will provide the best overall solution.

With the threat of IoT malware increasing both in number [6] and with new highly dangerous variants such as P2P malware [29], it is essential to have the best possible defence options. One of these options is the Moving Target Defense which has proven to be a promising defence strategy for IoT devices, especially when it includes a cooperative component, as shown in this thesis. To be prepared for future threats and especially against P2P malware, further research on how cooperative Moving Target Defense can defend against IoT malware is essential.

7.1 Future Research

As the topic of MTD is large and complex, there are several possibilities for further research. This thesis has shown that the cooperative and reactive MTD approach offers several advantages over other approaches. Examples of these advantages are a comparatively lower duration of infection activity on the system and a comparatively lower interruption of availability. A prerequisite for a reactive MTD approach to work is the ability to detect potential malware on the system. Therefore, research on fast and reliable malware detection mechanisms is essential. One possible means for that are machine learning algorithms [35].

Another research option is to further develop the presented solution. Possible improvements are to make the solution faster, for example by optimizing the *nmap* scan, or to adapt the solution for other operating systems. A significant research would be to investigate if and to what extent the current solution works against malware such as Mirai or P2P botnets. Theoretically, moving the Telnet service port should also protect against these malwares, but this should be tested in practice.

Another interesting area of research would be to further investigate the impact of the LAN setting. This has only been touched on slightly in this thesis, but the LAN setting could provide additional options for the malware that could have a negative impact on an MTD defence mechanism. For example, one of these potential options are port scanners that might be able to find the moved Telnet port.

Finally, the presented solution uses a classic client-server architecture. This was due to the simplicity provided, which allowed the basic concept of cooperative MTD to be tested more quickly. For a more sophisticated implementation, other architectures should be considered. For example, one option is a P2P architecture, which would provide the corresponding advantages, such as e.g. the removal of the single point of failure.

Bibliography

[1] Fortune Business Insights. "Internet of things (iot) market size, share covid-19 impact analysis, by component (platform, solution services), by end-use industry (bfsi, retail, government, healthcare, manufacturing, agriculture, sustainable energy, transportation, it telecom, and others), and regional forecast, 2022-2029". https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307 (accessed Sep. 18, 2022).

[2] Statista. "Number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030". https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/#:~:text= The number of Internet of Things(IoT) devices,in China with around 5 billion consumer devices (accessed Sep. 18, 2022).

[3] S. Agarwal, P. Oser, H. Short, and S. Lueders, "Internet of Things (IoT) security", Geneva, Tech. Rep., 2017. DOI: 10.5281/zenodo.1035034. [Online]. Available: https://cds.cern.ch/record/2776796.

[4] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets", *Computer*, vol. 50, no. 7, pp. 80–84, 2017. DOI: 10.1109/MC.2017.201.

[5] J. S. Perry. "Anatomy of an iot malware attack". https://developer.ibm.com/articles/iot-anatomy-iot-malware-attack/ (accessed Oct. 21, 2022).

[6] D. Demeter, M. Preuss, Y. Shmelev. "Iot: A malware story". https://securelist.com/iot-a-malware-story/94451/ (accessed Oct. 19, 2022).

[7] R. E. Navas, F. Cuppens, N. Boulahia Cuppens, L. Toutain, and G. Z. Papadopoulos, "Mtd, where art thou? a systematic review of moving target defense techniques for iot", *IEEE Internet of Things Journal*, vol. 8, no. 10, pp. 7818–7832, 2021. DOI: 10.1109/JIOT.2020.3040358.

[8] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques", *IEEE Security & Privacy*, vol. 12, no. 2, pp. 16–26, 2014. DOI: 10.1109/MSP.2013.137.

[9] A. Marzano, D. Alexander, O. Fonseca, *et al.*, "The evolution of bashlite and mirai iot botnets", in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00 813–00 818. DOI: 10.1109/ISCC.2018.8538636.

[10] J. Cedeño, "Mitigating cyberattacks affecting resource-constrained devices through moving target defense (mtd) mechanisms", 2022.

[11] A.K. Ghosh, I. D. Pendarakis, W.H. Sanders. "National cyber leap year summit 2009". https://www.nitrd.gov/nitrdgroups/images/b/bd/National_Cyber_Leap_Year_Summit_2009_CoChairs_Report.pdf (accessed Aug. 19, 2022).

[12] IBM. "What is an attack surface?" https://www.ibm.com/topics/attack-surface (accessed Jan. 27, 2023).

[13] G.-l. Cai, B.-s. Wang, W. Hu, and T.-z. Wang, "Moving target defense: State of the art and characteristics", *Frontiers of Information Technology Electronic Engineering*, vol. 17, pp. 1122–1153, 2016. DOI: 10.1631/FITEE.1601321.

[14] S. Santra and P. P. Acharjya, "A study and analysis on computer network topology for data communication", *International Journal of Emerging Technology and Advanced Engineering*, vol. 3, no. 1, 2013.

[15] gfu Consumer Home Electronics. "Smart homes: Survey in germany and the united kingdom". https://gfu.de/en/smart-home-befragung-in-deutschland-und-grossbritannien/ (accessed Sep. 18, 2022).

[16] M. Lombardi, F. Pascale, D. Santaniello, "Internet of things: A general overview between architectures, protocols and applications", *Information*, vol. 12, no. 2, 2021.

[17] Cloudflare. "What is a personal area network (pan)?" https://www.cloudflare.com/learning/network-layer/what-is-a-personal-area-network/ (accessed Jan. 07, 2023).

[18] Eclipse Foundation. "Iot & edge developer survey report". https://5413615.fs1.hubspotusercontent-na1.net/hubfs/5413615/2022 IoT & Edge Developer Survey Report.pdf?hsCtaTracking=005f9ab7-5a8b-4efb-8733-ccfc469871fa%7C49968bbc-8827-4c1d-87cb-836dd8419b3b (accessed Oct. 07, 2022).

[19] freeRTOS. "Freertos". https://freertos.org/ (accessed Oct. 07, 2022).

[20] Malwarebytes. "Malware". https://www.malwarebytes.com/malware (accessed Oct. 21, 2022).

[21] Q.-D. Ngo, H.-T. Nguyen, V.-H. Le, and D.-H. Nguyen, "A survey of iot malware and detection methods based on static features", *ICT Express*, vol. 6, no. 4, pp. 280–286, 2020, ISSN: 2405-9595. DOI: 10.1016/j.icte.2020.04.005.

[22] M. Vicente, B. Gelera, A. Remillano, C. Toyama, J. Urbanec. "Bashlite updated with mining and backdoor commands". https://www.trendmicro.com/en_us/research/19/d/bashlite-iot-malware-updated-with-mining-and-backdoor-commands-targets-wemo-devices.html (accessed Oct. 20, 2022).

[23] The Open Group. "Ieee std 1003.1™-2017". https://pubs.opengroup.org/onlinepubs/9699919799/ (accessed Oct. 19, 2022).

[24] F. Hofmann. "Is linux posix-compliant?" https://linuxhint.com/is_linux_posix_compliant/ (accessed Oct. 19, 2022).

[25] Comodo Antivirus. "Sophisticated linux trojan self-deletes to elude detection?" https://antivirus.comodo.com/blog/computer-safety/linux-trojan-self-deletes-elude-detection/ (accessed Oct. 20, 2022).

[26] Security Cam Center. "Dahua default password". https://securitycamcenter.com/dahua-default-password/ (accessed Oct. 20, 2022).

[27] M. R. Fuentes, S. Hilt, R. McArdle, F. Merces, and D. Sancho. "The future of p2p iot botnets". https://www.trendmicro.com/en_us/research/21/c/the-future-of-p2p-iot-botnets.html#:~:text=For%20P2P%20IoT%20botnets%20to,a%20mere%20internet%2Dconnected%20device (accessed Dec. 19, 2022).

[28] Europol. "Andromeda botnet dismantled in international cyber operation". https://www.europol.europa.eu/media-press/newsroom/news/andromeda-botnet-dismantled-in-international-cyber-operation (accessed Dec. 19, 2022).

[29] S. Hilt, R. McArdle, F. Merces, M. Rosario, and D. Sancho. "Uncleanable and unkillable: The evolution of iot botnets through p2p networking". https://documents.trendmicro.com/assets/pdf/Technical_Brief_Uncleanable_and_Unkillable_The_Evolution_of_IoT_Botnets_Through_P2P_Networking.pdf (accessed Dec. 19, 2022).

[30] G. Zhang and M. Parashar, "Cooperative defense against network attacks", Jan. 2005, pp. 113–122.

[31] B. Rodrigues, T. Bocek, A. Lareida, D. Hausheer, S. Rafati, and B. Stiller, "A blockchain-based architecture for collaborative ddos mitigation with smart contracts", 2017, pp. 16–29, ISBN: 978-3-319-60773-3. DOI: 10.1007/978-3-319-60774-0_2.

[32] B. Rodrigues, L. Eisenring, E. Scheid, T. Bocek, and B. Stiller, "Evaluating a blockchain-based cooperative defense", in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 533–538.

[33] Cloudflare. "What is an autonomous system? | what are asns?" https://www.cloudflare.com/learning/network-layer/what-is-an-autonomous-system/ (accessed Oct. 11, 2022).

[34] B. C. Ward, S. R. Gomez, R. W. Skowyra, D. Bigelow, J. N. Martin, J. W. Landry, H. Okhravi. "Survey of cyber moving targets second edition". http://web.mit.edu/br26972/www/pubs/mt_survey.pdf (accessed Oct. 05, 2022).

[35] J. von der Assen, A. Huertas, P. M. Sánchez, *et al.*, "A lightweight moving target defense framework for multi-purpose malware affecting iot devices", 2022. DOI: 10.48550/arXiv.2210.07719.

[36] A. A. Mercado-Velázquez, P. J. Escamilla-Ambrosio, and F. Ortiz-Rodríguez, "A moving target defense strategy for internet of things cybersecurity", *IEEE Access*, vol. 9, pp. 118 406–118 418, 2021. DOI: 10.1109/ACCESS.2021.3107403.

[37] R. E. Navas, H. Sandaker, F. Cuppens, N. Cuppens, L. Toutain, and G. Z. Papadopoulos, "IANVS: A moving target defense framework for a resilient internet of things", in *2020 IEEE Symposium on Computers and Communications (ISCC)*, 2020, pp. 1–6. DOI: 10.1109/ISCC50000.2020.9219728.

[38] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront, "MT6D: A moving target ipv6 defense", in *2011 - MILCOM 2011 Military Communications Conference*, 2011, pp. 1321–1326. DOI: 10.1109/MILCOM.2011.6127486.

[39] M. Sherburne, R. Marchany, and J. Tront, "Implementing moving target ipv6 defense to secure 6lowpan in the internet of things and smart grid", in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, New York, NY, USA: Association for Computing Machinery, 2014, pp. 37–40, ISBN: 9781450328128. DOI: 10.1145/2602087.2602107. [Online]. Available: https://doi.org/10.1145/2602087.2602107.

[40] K. Zeitz, M. Cantrell, R. Marchany, and J. Tront, "Designing a micro-moving target ipv6 defense for the internet of things", in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2017, pp. 179–184.

[41] V. Casola, A. De Benedictis, and M. Albanese, "A moving target defense approach for protecting resource-constrained distributed devices", 2013, pp. 22–29. DOI: 10.1109/IRI.2013.6642449.

[42] A. Judmayer, J. Ullrich, G. Merzdovnik, A. G. Voyiatzis, and E. Weippl, "Lightweight address hopping for defending the ipv6 iot", in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, New York, NY, USA: Association for Computing Machinery, 2017, ISBN: 9781450352574. DOI: 10.1145/3098954.3098975. [Online]. Available: https://doi.org/10.1145/3098954.3098975.

[43] F. Nizzi, T. Pecorella, F. Esposito, L. Pierucci, and R. Fantacci, "Iot security via address shuffling: The easy way", *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 3764–3774, 2019. DOI: 10.1109/JIOT.2019.2892003.

[44] VirtualBox. "Virtualbox". https://www.virtualbox.org/ (accessed Oct. 30, 2022).

[45] Nathan Jennings. "Socket programming in python (guide)". https://realpython.com/python-sockets/ (accessed Oct. 30, 2022)].

[46] F. Ding. "Iot malware". https://github.com/ifding/iot-malware (accessed Nov. 24, 2022).

[47] jgamblin. "Mirai-source-code". https://github.com/jgamblin/Mirai-Source-Code (accessed Nov. 24, 2022).

[48] hammerzeit. "Bashlite". https://github.com/hammerzeit/BASHLITE (accessed Nov. 24, 2022).

[49] Incredibuild. "Gcc". https://www.incredibuild.com/integrations/gcc (accessed Nov. 24, 2022).

[50] M. Kerrisk. "Fork(2) — linux manual page". https://man7.org/linux/man-pages/man2/fork.2.html (accessed Nov. 24, 2022).

[51] Raspberry Pi. "Raspberry pi desktop for pc and mac". https://www.raspberrypi.com/software/raspberry-pi-desktop/ (accessed Nov. 24, 2022).

[52] D. Libes. "Expect(1) - linux man page". https://linux.die.net/man/1/expect (accessed Nov. 28, 2022).

[53] SecTools. "Netcat". https://sectools.org/tool/netcat/ (accessed Nov. 24, 2022).

[54] Intel. "Intel® core™ i7-6600u processor". https://www.overleaf.com/project/62ea340eebbfc33642a986f3 (accessed Dec. 24, 2022).

[55] S. Guide. "Port 2323 details". https://www.speedguide.net/port.php?port=2323 (accessed Nov. 30, 2022).

[56] P. S. Foundation. "Subprocess — subprocess management". https://docs.python.org/3/library/subprocess.html (accessed Dec. 25, 2022).

[57] M. Kerrisk. "ss(8) — linux manual page". https://man7.org/linux/man-pages/man8/ss.8.html (accessed Dec. 23, 2022).

[58] D. Manpages. "INETD(8)". https://manpages.debian.org/testing/inetutils-inetd/inetutils-inetd.8.en.html#HISTORY (accessed Dec. 23, 2022).

[59] Gnu. "GNU sed". https://www.gnu.org/software/sed/manual/sed.html (accessed Dec. 25, 2022).

[60] J.-H. Cho, D. P. Sharma, H. Alavizadeh, *et al.*, "Toward proactive, adaptive defense: A survey on moving target defense", 2019. DOI: 10.48550/ARXIV.1909.08092. [Online]. Available: https://arxiv.org/abs/1909.08092.

[61] nmap. "Timing and performance". https://nmap.org/book/man-performance.html (accessed Jan. 21, 2023).

Abbreviations

6LoWPAN	IPv6 over Low power Wireless Personal Area Network
MTD	Moving Target Defense
MP	Moving Parameter
OS	Operating System
C&C	Command and Control
DDoS	Distributed Denial of Service
P2P	Peer-to-Peer

Glossary

Moving Target Defense A cybersecurity paradigm that aims to change the attack surface of a system by altering its properties, such as the IP addresses or the data representation.

Moving Parameter The ”what” that should be changed as a part of the Moving Target Defense. An example is the value of the IP address.

Internet of Things Items such as sensors that are connected through a network and collect and exchange data.

Botnet A collection of bots that are remotely controlled and can be used to launch cyber attacks.

Bashlite A well-known IoT malware that was the basis for other well-known IoT malware such as Mirai. Aims to create a botnet of infected devices.

Mirai An IoT malware still prevalent today that, like Bashlite, aims to create a botnet of infected devices.

HEH One of the few existing P2P malware with the goal of creating a botnet of infected devices.

Honeypot A way to analyze malware as honeypots attract these malicious software.

List of Figures

List of Tables

Appendix A

Installation Guidelines

All the files for this thesis can be found at `https://github.com/stevna/MScThesis_coopMTD`. This includes the modified version of Bashlite, all files related to the MTD framework and techniques, the scripts for data collection, and the Jupyter notebooks for data analysis. There is also a link to download all three complete virtual machines.

The Wiki tab of this GitHub page contains detailed installation and execution instructions, including how to run Bashlite and what scripts need to be executed to start the infection and mitigation processes. It also includes instructions on how to insert the downloaded virtual machines into Oracle VM VirtualBox so that the virtual machines do not need to be recreated for future research.

Appendix B

Contents of the GitHub Repository

The provided GitHub repository has the following contents:

1. This thesis as a PDF.
2. A short summary of the thesis in German.
3. The modified server and client code of Bashlite.
4. All files related to the MTD framework and techniques of all 3 virtual machines.
5. The files and scripts to collect the data for the evaluation.
6. The files and scripts to analyze the data for the evaluation.
7. The Latex source code used for this report,
8. The source files of the graphics and tables used for this thesis which is a Microsoft PowerPoint document.
9. A SWITCHdrive link where all three virtual machines can be downloaded for the subsequent insertion into Oracle VM VirtualBox.