SAPIENZA
UNIVERSITÀ DI ROMA

Graph and Flow-based Distributed Detection and Mitigation of Botnet Attacks

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea Magistrale in Engineering in Computer Science

Candidate

Alessio Izzillo

ID number 1710580

Thesis Advisor

Prof. Alessandro Pellegrini

Academic Year 2019/2020

Thesis defended on 21 January 2021
in front of a Board of Examiners composed by:

Prof. Daniele Nardi (chairman)
Prof. Roberto Capobianco
Prof. Francesca Cuomo
Prof. Alessandro De Luca
Prof. Daniele Cono D'Elia
Prof. Riccardo Lazzeretti
Prof. Domenico Lembo
Prof. Alessandro Pellegrini
Prof. Leonardo Querzoni
Prof. Giuseppe Santucci

Graph and Flow-based Distributed Detection and Mitigation of Botnet Attacks
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: izzillo.1710580@studenti.uniroma1.it

Contents

Chapter 1

Introduction

Nowadays, many organizations are constantly victims of several security threats which may cause economic and reputation damages. Among the large variety of malware which affects several systems and online services, we can found software such as worms, viruses, spyware, trojans, key-loggers and botnets. Botnets are one of the most dangerous type of cyber-attacks, which are used in a variety of malicious campaigns such as email spam, financial theft, click fraud, distributed denial-of-service (DDoS) attacks for taking online services offline, and for committing cryptocurrency scams (using users' processing power to mine cryptocurrency) [36]. According to the Federal Bureau of Investigation, "*Botnets have caused over $9 billion in losses to U.S. victims and over $110 billion in losses globally. Approximately 500 million computers are infected globally each year, translating into 18 victims per second*" [16]. The first official recognized Botnet named "EarthLink Spammer" appeared in 2000: it was created to send phishing emails in large numbers, masked as communications from legitimate websites. Over 1.25 million malicious emails were sent to collect sensitive information. The botnet had downloaded viruses on victims' computers when they clicked on the links in the emails, and these viruses remotely fed the information to the sender. In 2016, there was one of the largest and most lucrative digital ad malware ever devised: Methbot. It acquired thousands of IP addresses with US-based ISPs. The operators first created more than 6000 domains and 250267 distinct URLs that appeared to be from premium publishers (such as ESPN and Vogue), and then, video ads from malicious advertisers were

posted on these websites which sent their bots “watch” around 30 million ads daily [46].

Over the years, the Botnet technology has evolved making the detection and mitigation of botnet attacks a very challenging problem. Many approaches have been proposed (see chapter 2): signature-based, anomaly-based, DNS-based and mining-based. As it will be discussed, each approach presents advantages and disadvantages. The goal of this thesis is to use a hybrid analysis that relies on data mining flow and graph based patterns which can identify malicious external hosts which communicate with our hosts on which a distributed application of an online service is deployed. The hybrid analysis of this approach is performed online in order to mitigate the attacks. The process starts by capturing, in kernel space, some fields of the packets by means of an *eBPF filter* and passing them to userspace for grouping them in "batches" of a certain size, on which the hybrid analysis is carried out (see chapter 4).

The eBPF filter in kernel space allows to inspect in real-time the packets in an unobtrusive and effective way. In fact, in addition to collecting packets characteristics, it is able to reject the packets coming from external hosts having IPs labeled as malicious by the userspace analysis (see chapter 3).

Chapter 2

Botnet Detection Techniques

2.1 What is a Botnet?

As the word suggests, "botnet" is the union of "bot" which indicates an autonomous program, and the word "net" (that is the diminutive of "network", representing an interconnection among devices called "hosts").

Nowadays, botnets are regarded as networks of malicious or compromised hosts used mainly to perform several types of attacks with the aim of stealing data, sending spam, denying services, or allowing the attacker to compromise additional devices and their connection to expand the botnet itself.

The network of a botnet is composed of infected hosts (Bots) which are running autonomous software controlled by a human (bot herder), via one or more controllers (bot masters). The type of communication used by the bot master for communicating with its bots is called "Command and Control" (C&C) that allows to send commands and to receive responses by the infected hosts [33].

Botnets use two main types of architectures:

1. The "Centralized Architecture" is structured so that the Botmaster communicates, through one or more C&C servers, directly with all the bots to give commands and receive responses. Generally, the servers use IRC, HTTP or POP3 protocols to communicate with the bots and to send them the command to perform a specific type of attack like DDoS attacks, spamming, spying, etc. The centrality of this type of botnet causes a weakness because if the C&C

Servers are disabled in some way or go down, then the botnet will cease to exist [33].

2. The "P2P Architecture", in contrast to the previous type of botnets, does not have a communication (within the botnet) fully dependent on the central C&C Servers. In this case, P2P botnets use publish/subscribe systems to communicate: a set of commands is defined in the P2P system, and all bots subscribe to this set. Each bot of the network is able to launch an attack by publishing a command on the P2P system and all the bots subscribed to that command will be able to see it [33].

In addition to the previous main types of architectures, it can be added a third hybrid type of architecture: it is composed by "servant bots" and "soldier bots". The first ones communicate among them through a P2P protocol and behave like a "Server community" which are responsible for controlling the "soldier bots" by sending commands and receiving responses from them [54].

2.2 Overview of botnet detection techniques

Over the years, Botnets have become a significant cyber threat, and several botnet detection techniques have emerged to overcome their spreading and mitigating the damages. Each of these techniques are able to recognize a suspicious pattern in the traffic of a host or performing an analysis of the packets content that, in the newer botnets, is very often encrypted.

2.2.1 Signature-based Detection

Signature-based detection relies on signatures of the Botnets. For example, Snort [48] is an open source intrusion detection system (IDS) that finds intrusions by monitoring network traffic and looking for signs which identify a specific part of the traffic as suspicious. As every IDS, Snort is set with some rules (obtained by transforming the payload information of network traffic) which allow an immediate detection and impossibility of false positives. The main drawback is that this

approach can be only used for the detection of well-known Botnets, making this technique mostly ineffective for unknown Botnets.

One of the most known signature-based and mining-based techniques is Rishi [23] that is based primarily on finding suspicious IRC nicknames, IRC servers or uncommon IRC server ports by analyzing IRC-based botnets traffic. The advantage of Rishi is that it allows to detect bots which use uncommon communication channels by performing an n-gram analysis and a scoring system. This feature makes it able to detect bots which are not detectable by other classical Intrusion Detection Systems. The disadvantage is that it cannot detect encrypted communications as non-IRC Botnets and, as said before, it needs known nickname patterns to work [18, 58, 55].

2.2.2 Anomaly-based Detection

The Anomaly-based botnet detection approach, in contrast to the previous type of approach, requires no apriori knowledge of bot signatures, botnet C&C protocols, and C&C server addresses. It relies on network traffic anomalies such as high network latency, high volumes of traffic, traffic on unusual ports, and unusual system behavior that could mean that some malicious bots are present in the network.

The Anomaly-based detection systems can be divided in: Network-based and Host-based [18, 3].

Network-based detection

This type of approach relies on monitoring network traffic to find Botnets. The monitoring performed on the traffic by this approach can recognize it as malicious by observing its behavior, patterns, response time, network load, and link characteristics.

Besides, the Network-based detection systems can perform an *active monitoring* or a *passive monitoring*.

In *active monitoring*, test packets are injected into network, servers or applications for measuring the reactions of network and detect malicious activities. Among all works which have used this kind of approach, an interesting one is Botprobe [26]: it injects well-crafted packets into the payload of network traffic with a client (that is under monitoring) for finding suspicious patterns in the network activity caused by

humans or bots. For example, it can participate to an IRC chatting session and can, at a certain time, recognize a suspicious predetermined pattern that usually is used to transmit command to non-humans bots by C&C Servers. The main advantage of active monitoring is the response time, because it needs inspecting at most one round of C&C interaction. The disadvantage of this technique is that it produces extra traffic (for the injection of additional packets) causing an overhead in the usual network traffic. Moreover, it is not easy to divide legitimate traffic from the artificially injected traffic for anomaly detection, and this might raise privacy issues and the disruption of the legitimate traffic.

In *passive monitoring*, specialized devices are used to analyze network traffic. In contrast to active monitoring, in this case we have no increase of the traffic on the network for inspection (because there is no injection of additional test packets). One research that uses this approach is BotSniffer [28]: it is able to detect some response similarities of bots belonging to the same Botnet by performing an analysis allowing to detect spatial-temporal correlation in network traffic with a very low false positive rate. Instead, the research work of Binkley and Singh [6] uses the passive monitoring approach by combining TCP-based anomaly detection with IRC tokenization and IRC message statistics for detecting client Botnets and even for revealing bot servers. The drawback is that it requires long time to detect Botnets because it needs multiple rounds of Botnet communications and activities to inspect. Another drawback is that passive monitoring requires polling to collect data causing an increase of the network payload. This increase will be enormous if the device captures each packet for flow analysis; thus, it also causes security and privacy issues [22, 35, 58].

Host-based detection

Host-based detection systems perform the monitoring and the analysis of the internals of the computer system instead of network traffic (as it was the case in Network-based detection systems). This method aims to check whether the individual machine is infected by the bot or not by analyzing if bots have changed registry structure, system calls, and system files of the infected machine. Among

host-based tracking systems, we can find BotSwat [53]: it is a tool for monitoring home operating systems (such as Windows XP, Windows 2000, and Windows 7). Initially, it acts as a scanner which monitors the execution status of the Win32 library and observes run-time system calls created by the processor. Furthermore, it analyzes the received network data from unreliable external sources and tries to identify the remote behaviour of bots despite the particular C&C architecture, communication protocols, or botnet structure. A drawback of this approach is the lack of security for system calls. In addition, Botswat may cause non-negligible false-positives and performance penalty [58, 35]. Liu et al. (2008) proposed BotTracer [39]: this approach is able to detect the three life cycle phases of a botnet (startup, preparation, attack). It attempts to discover the channels used by the bots to connect with tge C&C network and after capturing them, it compares those channels with the known properties of the C&C channels on which the botnet traffic is moving. The disadvantage of BotTracer is the inability to detect the existence of virtual machines; however, it is able to monitor vulnerabilities in the system calls for any potential botnet activity. One advantage of using a host-based technique is that it can easily prevent "download attacks" and attacks attempting at start-up. A host-based approach characterized by this advantage, providing a real-time protection of the system, is that proposed by Xu *et al.*, in 2011, with DeWare [57]: a security tool that enforces the dependencies between user actions and system events, such as file-system access and process execution. This method, however, presents a security concerns caused because user-level OS routines may be intercepted with kernel-level routines, which may cause the OS to malfunction [35].

2.2.3 DNS-based Detection

DNS-based detection systems rely on the analysis of DNS information, originated by the botnet: the bots usually start a connection with a C&C Server to get commands. In order to establish a connection with the C&C Server, the bots perform DNS queries to get the location of the Server that typically hosted by a DDNS provider. In this phase, a DNS-based detection system analyzes DNS traffic and tries to detect possible communication anomalies. There are some studies which

use this approach to perform botnet detection relying on the "group activity" property of botnet DNS traffic and using DNS redirection to monitor botnets. However, they are easily evaded when a botmaster knows them that can disrupt this scheme by applying massive fake DNS queries which leads to the creation of a number of false alarms[18, 35].

An interesting study proposed by Choi *et al.* [11], in 2007, shows a particular DNS-based technique that works regardless of the type of bot and botnet by using an anomaly-based detection mechanism. This technique uses information of IP headers and allows to detect botnets even though they uses channel encryption methods (like SSH). In addition, it is able to detect also C&C Server migration because often botnets change its C&C Server.

A research of Cranor *et al.* (2001) [13] have proposed a method that uses a directed graph to trace DNS flows for identifying agents including clients, DNS servers, and authoritative roots involved in DNS service provisioning. In this graph the nodes represent the IP addresses of the DNS server machines and the edges represent queries generally originated by clients. This technique is based on large-scale trace analysis, and it is able to identify correctly those hosts involved in the DNS based botnet communication [35].

Another study of Dagon (2005) [15] shows a method to identify botnet C&C servers by detecting domain names with abnormally high or temporally concentrated DDNS query rates. This method is not very strong because it could easily be evaded by using faked DNS queries and, moreover, it produces many false positives caused by the misclassification of legitimate domains that use DNS with short TTL [18].

One of the most common disadvantages of the techniques which use the DNS-based approach is that it is possible to incur a large processing time in monitoring the vast network traffic [18].

2.2.4 Mining-based Detection

Mining-based detection approaches include several techniques which identify botnet C&C traffic by means of several data mining techniques like regression, classification, and clustering. The mining techniques can be applied to features

extracted from the received traffic. The main two groups of feature extraction approaches used in the literature are: *Flow-based* and *Graph-based*.

Flow-based detection systems rely on the concept of "flow" that can be defined in the following way: "*if two different packets have the same source/destination host/port and the same protocol, they belong to the same flow*" [10]. So, the first stage in this type of systems is to assign each received packet to a specific flow identified with a "flow ID" and then to extract some features from packets belonging to the same flow. These features are based on information present in the header of the packets like timestamp, size, protocol. The header information of these packets belonging to a certain captured flow are then aggregated to form the "flow features" like Average Inter-Arrival-Time of the flow, Average size packet in the flow and so on. Masud et al. (2008) [40] proposed an effective host-based botnet detection technique that consists in mining the traffic flows obtained from multiple log files installed on the host machines. This technique takes advantage of the fact that the the bots, typically, respond more quickly than humans. It can be efficiently performed for both IRC and non-IRC bots, because it does not impose any restriction on the botnet communication protocol; moreover, it is payload-agnostic, therefore, it is effective even if the C&C payload is encrypted or is not available [35, 18].

Instead, *graph-based* detection systems rely on "graph-based features", derived from flow-level information, which reflect the true structure of communications of hosts. This type of approach attempts to create a graph from the captured traffic where the IP addresses are the nodes and individual packets are directed edges [51]. For each node we can extract several features like: out-degree, in-degree, out-neighbors, in-neighbors, PageRank centrality, betweeness centrality, eigenvector centrality, authority and hub centralities, and local clustering coefficient. The drawback of graph-based detection systems (compared to flow-based detection systems) is the sheer amount of time it takes to extract features that it can be very long sometimes, but "*the most of the graph-based feature algorithms are easily parallelizable, so making use of more cores would result in speed improvements*" [51]. Instead, the disadvantage of using the Flows features is that they are often characteristic of specific protocol-based botnets, and are not generalizable to newer

botnet varieties [51]. On the contrary, graph features ignore information data from the packets, and focus on the topological structure of the communication among hosts using centrality-based graph measures. A recent work of Daya *et al.* (2020) [1] have propose a robust method named 'BotChase': composed by a two-phased graph-based bot detection system that leverages both unsupervised and supervised machine learning. The first phase is intended to prune presumable benign hosts, while the second phase performs an analysis to find with high precision bot hosts among the remaining pruned host.

Another type of classification within mining-based detection systems is based on the machine learning method used on the extracted features. The two main methods are: *supervised learning* or *unsupervised learning*.

Supervised learning techniques use labeled training datasets to create models. It is employed to learn and identify patterns in the known training data. Some of supervised learning methods used in Botnet detection are: Support Vector Machines, linear regression, naive Bayes, decision trees, k-nearest neighbor algorithm and Neural Networks (Multilayer perceptron). In particular, the Random forest algorithm (as shown in [10]) selects variables automatically during the model formation and establishes the optimal discriminant model achieving a high detection rate having a low false positive and false negative rates.

On the other hand, unsupervised learning uses unlabeled training datasets to create models which discriminate between patterns in the data. Some of unsupervised learning methods used in Botnet detection are: Self-Organizing Map and k-Means [1]. A well-known work of Gu *et al.* (2008) using an unsupervised learning method is Botminer [27]: it clusters similar communication traffic and similar malicious traffic; then, it identifies the hosts that share both similar communication patterns and similar malicious activity patterns by performing cross cluster correlation. The main advantage of this method is that it is not affected by protocol and structure of the botnets and can detect real-world Botnets (including IRC-based, HTTP-based, and P2P Botnets) with a very low false positive rate [58].

2.3 The Proposed Botnet Detection Approach

The method proposed in this work relies on a distributed system which allows to broadcast the host-captured traffic among the hosts belonging to that network such that each host knows the traffic of any other host of the network and can exploit this knowledge to improve the detection of Botnet attacks. Besides, it uses a hybrid mining-based detection technique resulting in an analysis which is divided in two parts: *flow-based* analysis and *graph-based* analysis.

Several studies in the literature have used hybrid methods and among them one of the most important is Botmark [56]: this automated model is based on the extraction of 15 flow-based features and 3 graph-based features to analyze the behaviour of each host present in the capture traffic. After performing the *flow-based* analysis (implemented with a "similarity-based algorithm" applied to the *flow-based* features), *graph-based* analysis, and *stability-based* analysis (that measures the stability of the flow, since the botnets are characterized by a relatively stability of the packets length distribution), the results of these three detectors are ensembled to mark a host as "bot" or "normal". More specifically, if more than two of these analyses identify the specific host as a bot, the host is then marked as a bot. Otherwise, it will be marked as normal.

The main difference of this proposed work with Botmark is the concept of "distributed network traffic capture" and the use of incremental learning. In the proposed work, the *flow-based* analysis is cyclically performed on a batch of captured packets with a predetermined size, whereas the *graph-based* method performs the analysis both on locally captured packets and shared packets which other hosts of the distributed system have broadcasted. The sharing by each host of the own traffic with other hosts of the distributed system allows each of them to construct a graph with much more information that can lead each host to identify a malicious pattern of an external host not only through the traffic exchanged with it (that can make the *graph-based* method inaccurate for several types of attack) but also with the "shared knowledge" identified by the traffic captured and broadcast by other hosts of the distributed system. *Flow-Based* analysis is more responsive than *graph-based* one (due to the lower amount of traffic to analyze) but it is less accurate at first.

The accuracy of the *flow-based* method will increase over time thanks to incremental learning, that will improve and expand the training dataset of the *flow-based* method by relying on the predictions of the *graph-based* method.

Chapter 3

Real-Time Packet Inspection

Monitoring the incoming and outgoing network traffic of a hosts is an important element in computer networks. Over the time, the implementations designated for this purpose revealed some problems:

- **Overhead**: caused by the slowness in filtering traffic.
- **Awkward policy setting**: packet filtering rules should be shipped by application developers alongside their product, aiming to secure-by-default systems while allowing techniques like port-knocking without root access.
- **Centralized ruleset**: the management of huge amounts of configuration files that complicates the central policy implementation and verification. Policy rules become simpler if only one application on a host at a time is considered; moreover, a single unit composed by packet filtering and the actual application, may avoid to unintentionally expose that application to the network.
- **Performance**: number of packets processed per second (Mbps) in relation to the number of rules to apply.

In the past, the main solutions were based on a set of hosts' packet filtering rules installed in a specific location of the kernel. However, the old implementation of kernel space filtering caused overhead since the received packet needed to be copied to memory and to undergo basic processing. To overcome these problems, without the need to use hardware support, it is possible to use a revolutionary technology

named "eBPF". This technology allows to add application-specific eBPF-based packet filters to sockets by using `systemd`'s socket activation allowing to set some rules only for a specific application and simplifying the central system-level firewall. [49]

3.1 eBPF

Traditionally packet filtering has been carried out through mechanisms running in userspace, which meant that each received packets needed to be copied from Kernel space before being filtered.

In 1992, a new technology for packet filtering appeared in a paper titled "The BSD Packet Filter: *A New Architecture for User-Level Packet Capture*" [41]. This paper described a new technology, named *Berkeley Packet Filter* (BPF), for filtering network packet in Unix-like systems 20 times faster than the state of the art in packet filtering at the time. BPF success was based on two big innovations:

- A new virtual machine (VM) designed to work efficiently with register-based CPUs.
- The usage of per-application buffers that could filter packets without copying all the packet information. This minimized the amount of data BPF required to make decisions.

As a result, these new features improved packet filtering performance and all Unix systems started to adopt BPF.

In September 2013, Alexei Starovoitov proposed a patchset named “extended BPF” and in December 2013, Alexei was already proposing its use for tracing filters. While it was still a proposal, Daniel Borkmann (a kernel engineer at Red Hat) helped to include the new version of BPF in the Kernel and to replace the existing version. Alexei and Daniel saw the patches that began to merge in the Linux kernel by March 2014. The introduction of the BPF extension has led to rename the original BPF as *classic BPF* (cBPF) to distinguish it from the new implementation, the *extended BPF* (eBPF). The initial goal for this new implementation was to optimize the internal BPF instruction set that processed network filters. At this

point, BPF was still restricted to kernel-space, and only a few programs in user-space could write BPF filters for the kernel to process, like *tcpdump* and *seccomp*. Today, these programs still generate bytecode for the old BPF interpreter, but the kernel translates those instructions to the much improved internal representation. In June 2014, JIT components were merged in Linux 3.15, and, in December 2014, the `bpf()` syscall for controlling BPF was merged in Linux 3.18, allowing eBPF to be exposed to user-space. Later additions in the Linux 4.x series added BPF support for *kprobes*, *uprobes*, *tracepoints*, and *perf_events*.

eBPF also increased the number of registers and their size from two 32-bit registers to ten 64-bit registers. This improvement with other new features allowed the developers to write more complex programs and made the extended BPF version up to four times faster than the original BPF implementation. The BPF programs seem to work similarly to Kernel modules but with a better emphasis on safety and stability. [9]

3.1.1 eBPF Internals

As we can see in Figure 3.1, the eBPF Subsystem is constituted by several parts.

The source code of a filter can be written in C and then compiled using LLVM (or also GCC) allowing us to obtain a BPF bytecode that can be executed in the kernel.

After the compilation, a second C program is used to load the previously eBPF compiled program to kernel through the `bpf()` syscall. For loading a program, we have to invoke `bpf(BPF_PROG_LOAD, ...)` passing the BPF program as parameter within the user attributes union `bpf_attr __user *uattr`. The handler of the syscall will call `bpf_prog_load()`, and this function will load the eBPF program from userspace, and will then invoke the eBPF Verifier passing it the eBPF instructions to execute.

The Verifier ensures that the eBPF program is safe to run by the kernel and prevents to run code that might compromise the system (e.g. kernel crash).

If the eBPF instructions, are considered safe by the compiler, then the eBPF program can be: either compiled through a *just-in-time* (JIT) compiler for eBPF

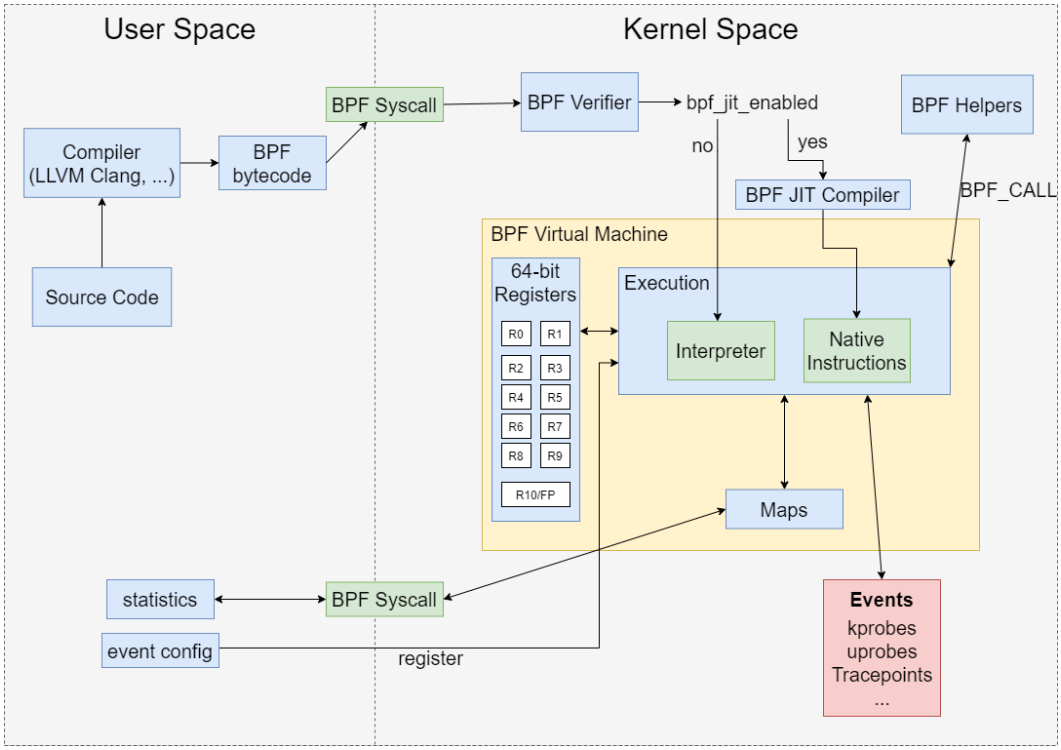


Figure 3.1. eBPF internals

instructions which will transform eBPF bytecode into machine code avoiding overhead at execution time, or passed to the "eBPF Interpreter" where it will be executed with a bit of overhead.

Before the execution of the eBPF program, we must specify the execution point in the kernel the program is attached to. We can associate the eBPF program to several type of mechanisms for monitoring events inside the system: sockets, kprobes, uprobes, tracepoints, user statically defined tracepoints (usdt).

The eBPF program can invoke a set of in-kernel functions called "helpers" associated with its specific program type to work with the data that it receives, to interact with the system or with eBPF maps, structures which are bidirectional (i.e. they can be accessed from both sides, the kernel and userspace) to share data with userspace. [25]

3.1.2 eBPF Registers

As we can see from Table 3.1, more registers and more storage space were introduced into the eBPF architecture with respect to the "classic" version, and the register size switched from 32-bit to 64-bit.

The 64 bit registers have 32-bit subregisters. The operating mode is 64 bit by default, the 32 bit subregisters can only be accessed through special ALU (arithmetic logic unit) operations. The 32-bit lower subregisters zero-extend into 64 bits when they are being written to. 32-bit architectures run 64-bit eBPF programs via interpreter. Their JITs may convert BPF programs that only use 32-bit subregisters into the native instruction set and let the rest being interpreted.

An eBPF program running in the the Kernel can call other in-kernel *helper* functions. These functions are invoked by an eBPF program through the following calling convention:

- R0: contains the return value from the helper function.
- R1 - R5: contain arguments to the helper function
- R6 - R9: they are callee-save registers that helper function will preserve
- R10: it is a read-only register that contains the frame pointer to access the eBPF stack.

	Classic BPF	**Extended BPF**
Registers	accumulator A, index register X	R0 - R9, plus a read-only frame pointer
Register size	32 bits	64 bits
Storage	16 memory slots: M[0-15]	512 bytes of stack space, plus infinite "map" storage

Table 3.1. Comparison between cBPF and eBPF

This calling convention restricts the number of arguments from an eBPF program to a helper in-kernel function to 5, and one register is used to return values from the

R0	rax
R1	rdi
R2	rsi
R3	rdx
R4	rcx
R5	r8
R6	rbx
R7	r13
R8	r14
R9	r15
R10	rbp

Table 3.2. Mapping of the eBPF registers to the x64 registers

helper function. The register R0 is also the register containing the exit value for the BPF program.

The registers R1 - R5 are "scratch registers", meaning that they are not preserved across a helper function call, so they need the caller function (the eBPF program) to either copy their content to the eBPF stack or to callee saved registers if the values need to be reused.

Moreover, the eBPF calling convention maps directly to ABIs used by the kernel on x86_64-bit architectures. For example, when the eBPF bytecode is passed to the x64 JIT compiler (to translate it into native instructions), it will map one to one the eBPF registers to the x64 registers as in Table 3.2.

In this way, the JIT compiler doesn't need to emit extra move instructions causing additional performance penalties. The function arguments will be in the correct registers and the `BPF_CALL` instruction will be translated by the JIT compiler to a `call` instruction. [50, 12]

3.1.3 eBPF Instruction set

eBPF reused the opcode encoding of cBPF to simplify conversion of cBPF programs to eBPF programs. Nowadays, the Linux kernel runs eBPF programs only, so the cBPF bytecode is transparently translated into an eBPF representation in the kernel and then executed in an interpreter or compiled by the JIT compiler to run as native machine code.

The maximum instruction limit per program is restricted to 4096 BPF instructions, which, by design, means that any program will terminate quickly. For kernel newer than 5.1 this limit was lifted to 1 million eBPF instructions.

The eBPF instruction format is modeled as two operand instructions, that makes easier to map eBPF instructions to native instructions during the JIT compiling phase.

The instruction encoding of a single 64 bit instruction is shown in Figure 3.2. The element `op` is a 16 bit wide opcode that has a particular instruction encoded. For arithmetic and jump instructions, the 8-bit `op` field is divided as shown in Figure 3.3: where `instruction class` is the more generic instruction class (see Table 3.3), `operation code` denotes a specific operational code inside that class (see Table 3.4 and Table 3.5), and `source` specifies whether the source operand is a register (1) or an immediate value (0). Instead, for load and store instructions, the 8-bit `code` field is divided as shown in Figure 3.4: where `size` indicates the *size modifier* which encodes size of the operation (see Table 3.6), `mode` indicates the *mode modifier* which encodes how performs the operation (see Table 3.7). Both `dst_reg` and `src_reg` correspond to the destination and source register operands (e.g. R0 - R9), respectively, to be used for the operation. `off` is used to provide a relative offset in some types of instructions, for example: for addressing the stack or other buffers available to BPF (e.g. map values, packet data, etc), or jump targets in jump instructions. `imm` contains an immediate value.

An important difference between cBPF and eBPF regards the last two `instruction class` codes in Table 3.3. In fact, unlike eBPF, cBPF uses the following `instruction class`: `BPF_RET` with code `0x06` and `BPF_MISC` with code `0x07`. So, cBPF wastes a whole instruction class (`BPF_RET`) to represent a single `ret` operation, instead, eBPF

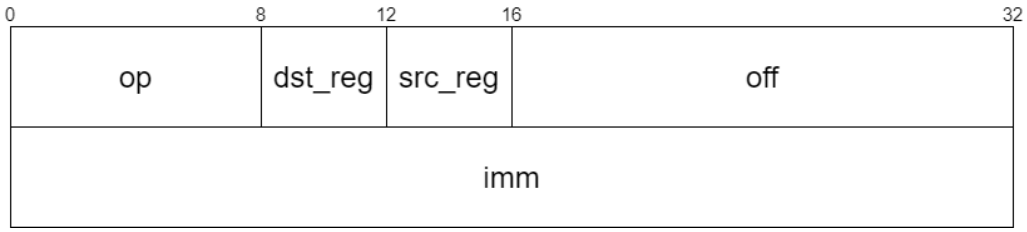


Figure 3.2. Instruction encoding of a 64 bit instruction

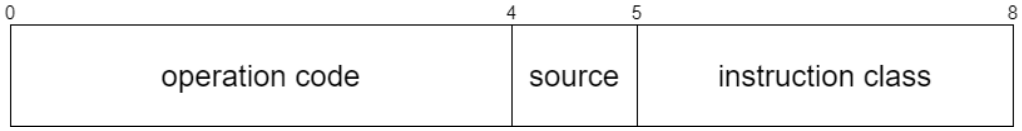


Figure 3.3. 'op' field of the arithmetic and jump instructions

Instruction class name	Instruction class code	Description
BPF_LD	0x00	load from immediate
BPF_LDX	0x01	load from register
BPF_ST	0x02	store immediate
BPF_STX	0x03	store value from register
BPF_ALU	0x04	32 bits arithmetic operations
BPF_JMP	0x05	jump operations
BPF_JMP32	0x06	unused, reserved for future use
BPF_ALU64	0x07	64 bits arithmetic operations

Table 3.3. Instruction classes

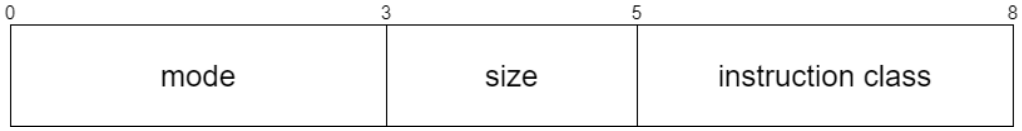


Figure 3.4. 'code' field of the load and store instructions

Operation name	Operation code	Description
BPF_ADD	0x00	
BPF_SUB	0x10	
BPF_MUL	0x20	
BPF_DIV	0x30	
BPF_OR	0x40	
BPF_AND	0x50	
BPF_LSH	0x60	
BPF_RSH	0x70	
BPF_NEG	0x80	
BPF_MOD	0x90	
BPF_XOR	0xa0	
BPF_MOV	0xb0	eBPF only: mov reg to reg
BPF_ARSH	0xc0	eBPF only: sign extending shift right
BPF_END	0xd0	eBPF only: endianness conversion

Table 3.4. operation codes for arithmetic instructions (corresponding to the instruction classes `BPF_ALU` or `BPF_ALU64`)

Operation name	Operation code	Description
BPF_JA	0x00	BPF_JMP only
BPF_JEQ	0x10	
BPF_JGT	0x20	
BPF_JGE	0x30	
BPF_JSET	0x40	
BPF_JNE	0x50	eBPF only: jump !=
BPF_JSGT	0x60	eBPF only: signed '>'
BPF_JSGE	0x70	eBPF only: signed '>='
BPF_CALL	0x80	eBPF BPF_JMP only: function call
BPF_EXIT	0x90	eBPF BPF_JMP only: function return
BPF_JLT	0xa0	eBPF only: unsigned '<'
BPF_JLE	0xb0	eBPF only: unsigned '<='
BPF_JSLT	0xc0	eBPF only: signed '<'
BPF_JSLE	0xd0	eBPF only: signed '<='

Table 3.5. operation codes for jump instructions (corresponding to the instruction classes `BPF_JMP` or `BPF_JMP32`)

Size modifier name	Size modifier code	Description
BPF_W	0x00	word
BPF_SUB	0x08	half word
BPF_MUL	0x10	byte
BPF_DIV	0x18	eBPF only, double word

Table 3.6. Size modifiers

operation name	operation code	description
BPF_IMM	0x00	used for 32-bit mov in classic BPF and 64-bit in eBPF
BPF_ABS	0x20	
BPF_IND	0x40	
BPF_MEM	0x60	
BPF_LEN	0x80	classic BPF only, reserved in eBPF
BPF_MSH	0xa0	classic BPF only, reserved in eBPF
BPF_XADD	0xc0	eBPF only, exclusive add

Table 3.7. Mode modifiers

performs this operation by using the `BPF_JMP instruction class` and setting the `operation code` to `BPF_EXIT`. In addition, the `BPF_MISC` class is substituted with `BPF_ALU64` to perform arithmetic operations in 64 bit mode. [50, 12]

3.2 Maps

eBPF Maps are the means by which eBPF programs and user-space programs can communicate with each other. Maps are efficient key/value stores that reside in kernel space and can be shared with other eBPF programs or user space applications. The Maps can be accessed by eBPF programs in order to keep state among multiple eBPF program invocations, or can be accessed by user space programs by using file descriptors. The eBPF Verifier includes several checks to ensure that the way used for creating and accessing maps is safe. [50, 12]

3.3 Network traffic capture

"Network traffic capture" or "Packet capture" is the process of intercepting a data packet that is crossing a specific point in a data network. The packets capture in real-time are stored for a period of time so that they can be analyzed, and then either be kept or discarded.

Our analysis is performed by an eBPF (see section 3.1) program written in C, loaded in Kernel Space and attached to a specific created socket through the Python

framework BCC[1] [29]. Captured packets are transferred to userspace for further analysis, as we shall discuss in the next chapter.

3.3.1 The eBPF Program

The eBPF filter has two purpose:

- Retrieving incoming and outgoing traffic to send packet information to userspace for the analysis (see chapter 4).
- Mitigating incoming and outgoing traffic with hosts considered 'suspicious' by the userspace analysis (as we shall discuss in section 4.5).

```
#include <bcc/proto.h>
#include <bcc/helpers.h>

struct Packet {
  u64 timestamp;
  u32 src_ip;
  u32 dst_ip;
  unsigned short src_port;
  unsigned short dst_port;
  unsigned int ethertype;
  unsigned char protocol;
  unsigned char tcp_Flags;
  unsigned short len;
  unsigned short tcp_payload_len;
  unsigned short udp_len;
  unsigned char ttl;
};

BPF_QUEUE(queue, struct Packet, 1024);
BPF_HASH(suspicious_IPs, u32, u32);
BPF_HASH(P2P_IPs, u32, u32);

int ebpf_program(struct __sk_buff *skb) {
```

[1] BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples.

```
24   struct Packet packet;
25   struct ethernet_t *ethernet = NULL;
26   struct ip_t *ip = NULL;
27   struct tcp_t *tcp = NULL;
28   struct udp_t *udp = NULL;
29   u8 *cursor = 0;
30
31   // ...
32 }
```

Listing 3.1. eBPF program structures

The Listing 3.1 shows the structures defined and initialized by an eBPF filter which are used for the first aforementioned objective. The `struct Packet` structure is used to keep information coming from the incoming/outgoing packet on the socket the eBPF Program is attached to. These `struct Packet` structures are enqueued in the map `queue` (in Listing 3.1 at line 19) that is a queue structure used to pass `struct Packet` structures to Userspace for the actual Botnet analysis. Moreover, as we will see in chapter 4, `suspicious_IPs` and `P2P_IPs` are hash maps used to pass the IPs of the hosts identified as "suspicious" and the IPs of the peers belonging to the P2P Network, respectively, to the eBPF filter.

In addition, as shown in Listing 3.3 at lines 25-28, pointers to BCC structures are initialized to point to header contents of the packet retrieved by means of the macro `cursor_advance()` (Listing 3.2). The fields of the packet header are then used to fill the fields of the `struct Packet` structure, as shown in Listing 3.3 at lines 8-43.

```
1 #define cursor_advance(_cursor, _len) \
2   ({ void *_tmp = _cursor; _cursor += _len; _tmp; })
```

Listing 3.2. cursor_advanced implementation

```
1 int ebpf_program(struct __sk_buff *skb) {
2   // ...
3
4   ethernet = cursor_advance(cursor, sizeof(*ethernet));
5
6   __builtin_memset(&packet, 0, sizeof(packet));
```

```
7
8   packet.ethertype = ethernet->type;
9
10  if (packet.ethertype == 2048){
11    ip = cursor_advance(cursor, sizeof(*ip));
12    packet.timestamp = bpf_ktime_get_ns();
13    packet.dst_ip = ip->dst;
14    packet.src_ip = ip->src;
15    packet.protocol = ip->nextp;
16    packet.len = ip->tlen+sizeof(*ethernet);
17    packet.ttl = ip->ttl;
18  }
19  else{
20    bpf_trace_printk("ALLOWED packet\n");
21    return -1;
22  }
23
24  if (packet.protocol == 6){
25    tcp = cursor_advance(cursor, sizeof(*tcp));
26    packet.src_port = tcp->src_port;
27    packet.dst_port = tcp->dst_port;
28    packet.tcp_Flags = 128*tcp->flag_cwr+64*tcp->flag_ece+32*tcp->
     flag_urg+16*tcp->flag_ack+8*tcp->flag_psh+4*tcp->flag_rst+2*tcp->
     flag_syn+1*tcp->flag_fin;
29    packet.tcp_payload_len = ip->tlen-(ip->hlen+tcp->offset*4);
30    packet.udp_len = 0;
31  }
32  else if (packet.protocol == 17){
33    udp = cursor_advance(cursor, sizeof(*udp));
34    packet.src_port = udp->sport;
35    packet.dst_port = udp->dport;
36    packet.tcp_Flags = 0;
37    packet.tcp_payload_len = 0;
38    packet.udp_len = udp->length;
39  }
40  else{
41    bpf_trace_printk("ALLOWED packet\n");
42    return -1;
```

```
43   }
44
45
46   if ((packet.protocol == 6) || (packet.protocol == 17)){
47     if (P2P_IPs.lookup(&packet.dst_ip) != NULL){
48       if ((packet.src_port == 9020) || (packet.dst_port == 9020) || (
     packet.src_port == 8000) || (packet.dst_port == 8000)){
49         bpf_trace_printk("ALLOWED packet: 'dst_ip' in 'P2P_IPs' hash
     and '(dst/src)_port' used in P2P network\n");
50         return -1;
51       }
52     }
53     else{
54       if (P2P_IPs.lookup(&packet.src_ip) != NULL){
55         if ((packet.src_port == 9020) || (packet.dst_port == 9020) ||
      (packet.src_port == 8000) || (packet.dst_port == 8000)){
56           bpf_trace_printk("ALLOWED packet: 'src_ip' in 'P2P_IPs'
     hash and '(dst/src)_port' used in P2P network\n");
57           return -1;
58         }
59       }
60     }
61
62     queue.push(&packet, BPF_EXIST);
63   }
64
65   // ...
66 }
```

Listing 3.3. Retrieving packet information and pushing them into BPF `queue` map

The first call to `cursor_advance()` (in Listing 3.3 at line 4) returns the point of the beginning of the packet that is assigned to `struct ethernet_t *ethernet`: this structure allows (through its attributes) to access the different fields of the Ethernet header. The `_cursor` was moved forward by the length of the Ethernet Header and a second call of `cursor_advance()` (in Listing 3.3 at line 11) allows to read the next header belonging to the IP layer. Finally, the third and final invocation of the macro `cursor_advance()` (in Listing 3.3 at lines 25,33) returns the pointer

to the L4 layer header (e.g. TCP, UDP).

The eBPF program considers (for the analysis) only IPv4 TCP or UDP packets, and all the other type of packets are not considered for the analysis (Listing 3.3, lines 10,24,32).

3.3.2 Loading the eBPF Program

The loading of the eBPF program is performed by a Python script and it is shown in Listing 3.4.

```
1  global bpf
2
3  # Initialize BPF - load source code from file
4  bpf = BPF(src_file=os.path.dirname(os.path.abspath(__file__))+ \
5      "/eBPF/eBPF_program.c", debug=0)
6
7  # ...
8
9  # Load eBPF program ebpf_program of type SOCKET_FILTER into the
10 # kernel eBPF vm.
11 function_ebpf_program = bpf.load_func("ebpf_program", \
12     BPF.SOCKET_FILTER)
13
14 # Create raw socket, bind it to interface and attach bpf program
15 # to socket created.
16 try:
17     BPF.attach_raw_socket(function_ebpf_program, interface)
18 except:
19     # ...
20
21 # Get file descriptor of the socket previously created inside
22 # BPF.attach_raw_socket.
23 socket_fd = function_ebpf_program.sock
24
25 # Create python socket object, from the file descriptor
26 sock = socket.fromfd(socket_fd, socket.PF_PACKET, socket.SOCK_RAW, \
27     socket.IPPROTO_IP)
28
```

```
29 # Set it as blocking socket
30 sock.setblocking(True)
31
32 # Get pointer to bpf map 'queue' of type 'BPF_QUEUE'
33 bpf_queue = bpf['queue']
34
35 bpf_hash_suspicious_IPs = bpf['suspicious_IPs']
36 bpf_hash_suspicious_IPs.clear()
```

Listing 3.4. Loading the eBPF Program

The `BPF` object is provided by the BCC framework and it allows to load an eBPF program from a C file through the `src_file` argument. Then, the function `ebpf_program` of the type `SOCKET_FILTER` into the C file is loaded into the kernel eBPF Virtual Machine.

From the `BPF` object is possible to create a raw socket binding it to an Ethernet interface and attaching the previously loaded eBPF program. Finally, the pointers of the BPF maps `queue` and `suspicious_IPs` are get (as shown at lines 33,35 of Listing 3.4).

3.3.3 Header packet information fetching

After initializing the BPF object and getting the BPF map pointers, the Python script executes an infinite while loop for fetching the `struct Packet` structures one by one from `queue` map, as shown in Listing 3.5.

```
1  from time import time
2
3  start_epoch = time()
4  start = None
5  while 1:
6      n = 0
7      while n < n_packets:
8          try:
9              # Get element by queue map
10             k = bpf_queue.pop()
11         except KeyError:
12             continue
```

```
13
14          if (start == None):
15              start = k.timestamp
16
17          # Compute the timestamp (in seconds) and add it to
18          # start Unix epoch
19          ts = (k.timestamp-start)/1000000000+start_epoch
20
21          # Update the Dataframe of the captured packets
22          Packets.loc[len(Packets)] = [ts, k.src_ip, \
23              k.dst_ip, k.src_port, k.dst_port, k.ethertype, \
24              k.protocol, k.tcp_Flags, k.len, \
25              k.tcp_payload_len, k.udp_len, k.ttl]
26
27          n += 1
28
29      # Enqueue in TaskQueue BotnetDetection_Threads a new
30      # BotnetDetection thread
31      BotnetDetection_threads.put(BotnetDetection(Packets.copy(), \
32          IncrementalLearning_threads)
33      Packets.drop(Packets.index, inplace=True)
```

Listing 3.5. Map elements fetching

At each loop, one `struct Packet` is retrieved from the `queue` map and the fields of the retrieved `struct Packet` structures are stored in the last line of the *DataFrame*[2] object `Packets`. After the DataFrame `Packets` reaches a certain size (argument `n_pkts`), a new task is created with the `BotnetDetection`[3] class and it is pushed in the `BotnetDetection_threads` (Listing 3.5, line 31), a `TaskQueue` structure.

`TaskQueue` is a Python subclass of `threading.Thread` whose purpose is to create a queue of threads to execute sequentially. The implementation (shown in Listing 3.6) is realized by defining a class variable `taskqueue`, that is initialized with a `Queue` object (of the Python library `queue`), and by using a while loop to pop, at each

[2] *DataFrame* is a Python object defined in the *pandas* library.

[3] The `BotnetDetection` class in Listing 3.5, corresponds to the *Botnet Detection Module* described in section 4.2.

iteration, a thread element from `taskqueue` and to execute it. The `TaskQueue` class is used to create:

- a queue `BotnetDetection_threads` composed of `BotnetDetection()` threads for performing the Flow-Based analysis (see section 4.2).
- a queue `IncrementalLearning_threads` composed of `IncrementalLearning()` threads for performing the Graph-Based analysis and updating the training dataset of the Flow-based analysis with the Graph-based analysis predictions (see section 4.3).

```
import threading
from queue import Queue

class TaskQueue(threading.Thread):
    def __init__(self):
        self.current_thread = None
        self.taskqueue = Queue()
        threading.Thread.__init__(self)

    def run(self):
        while not self._stopper.isSet():
            if self.taskqueue.empty():
                if self._stopper_when_empty.isSet():
                    break
                else:
                    continue
            self.current_thread = self.taskqueue.get()
            self.current_thread.start()
            self.current_thread.join()

    def put(self, thread):
        self.taskqueue.put(thread)
```

Listing 3.6. TaskQueue implementation

Chapter 4

The Detection and Mitigation Architecture

The architecture (shown in Figure 4.1) is composed by three main parts:

- The **eBPF filter**: it runs in kernel space with the purpose of retrieving incoming/outgoing packets, extracting their header information and passing them to the userspace *Botnet detection system*. At the end, the results of the botnet analysis performed by the userspace system are used to mitigate the network traffic from/to detected suspicious external hosts (see section 4.5).
- The **Botnet detection system**: it runs in userspace and analyzes the header packet information passed by the eBPF filter (by means of **queue QUEUE MAP**) attempting to detect malicious hosts belonging to some type of Botnet and returning the predictions resulted from the **Flow-based analysis** (executed through the **Botnet Detection module**) and the **Graph-based analysis** (executed through the **Incremental Learning Module**) to the eBPF filter.
  - **Flow-based analysis**: a type of analysis which relies on features extracted from the *flows* of network traffic captured by a host.
  - **Graph-based analysis**: a type of analysis based on constructing a graph which reflects the true structure of communications, interactions, and

behavior of the host towards external hosts, and on extracting by this graph some features which allow to determine the behaviour of each node (host) with the other nodes.

- The **P2P Network**: it is a collection of hosts connected to each other (by means of **P2P Module**) to allow the sharing of its own host-captured network traffic. This makes aware all hosts belonging to this network of the traffic captured by any other host in order to keep a complete knowledge to build a graph which is going to perform an effective botnet graph-based analysis.

The **Header packet information fetching** shown in the Figure 4.1 has been already explained in subsection 3.3.3. Instead, in the following sections, we will analyze in depth the remaining several parts which compose the architecture.

4.1 P2P Network

The **P2P module** relies on Twisted: an event-driven networking engine written in Python [19]. This module allows to start a "Peer to Peer" communication among hosts which run the tool. In this way, each host is able to broadcast the received header packet information of its own network traffic (acquired through the eBPF Program) to all hosts which belong to the P2P Network and, at same time, to receive from them their own network traffic. As a result, all peers know the whole network traffic of the P2P Network towards external hosts; in this way, each peer is able to build a graph based on this *global P2P traffic* and to extract several graph features from it. The P2P module starts by initializing a twisted Server which listen on the port 8000 for receiving `PING` and `PONG` messages in order to know active hosts connected to the P2P network.

A host takes part to the P2P Network by connecting to an active peer of the P2P Network. A `PING` message is sent by a host immediately after it is connected to a peer of the P2P Network. The peer to which the host connects, forwards this `PING` message to all the peers that it knows and decreases the *Time to live* field (`ttl`) of the `PING` message. Then, the peers which receive the `PING` message, in turn, forward it until the `ttl` is equal to 0. The structure of the `PING` message is

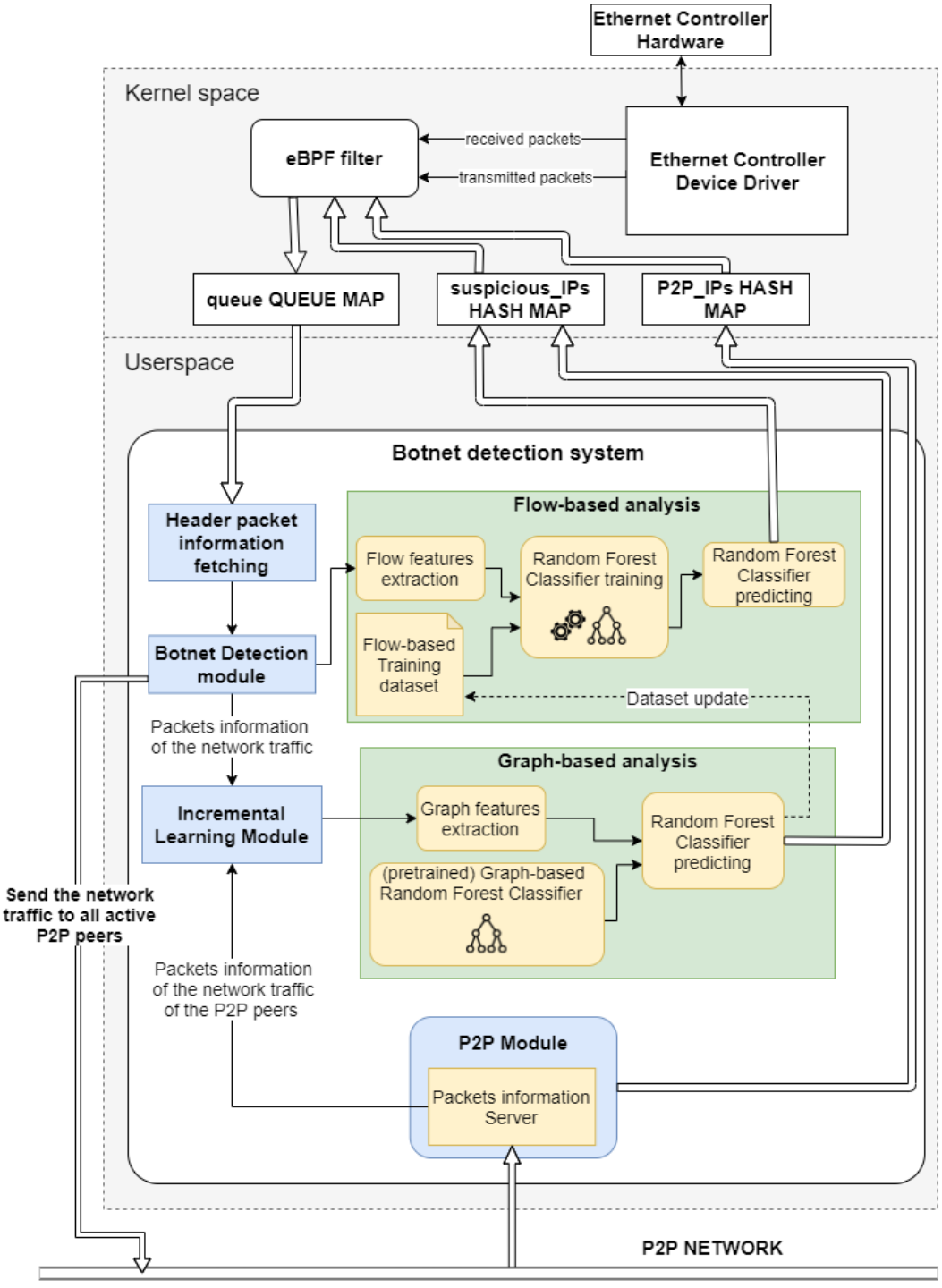


Figure 4.1. Botnet detection and mitigation architecture

the following: `<msgID>&00&<ttl>&<port>&<src_IP>`. In addition, the peers which receive a `PING` message from a new active peer, send the `src_IP` of the `PING` message to the eBPF filter. The IPs of the P2P Network peers are sent to the eBPF filter by storing them into **P2P_IPs HASH MAP** (see Figure 4.1); in this way, as shown in Listing 3.3, the network traffic exchanged among P2P peers will not be pushed in the **queue QUEUE MAP** (see Figure 4.1) and therefore it will not be considered in the botnet analysis. Moreover, the peers which receive the `PING` reply with a `PONG` message. The `PONG` message is a response message to the sender of the `PING` message, with the following structure: `<msgID>&01&<ttl>&<port>&<src_IP>`. When the peer that has sent the `PING` receives the `PONG` message, it sends the `src_IP` of the `PONG` message to the eBPF filter and forwards the `PONG` message to the other peers after decreasing the *Time to live* field (`ttl`) of the message. This technique allows each peer of the P2P Network to have a list (in **P2P_IPs HASH MAP**) of the IPs of all the others peers.

Once a host joins to the P2P Network, the **Packets information Server** starts listening on the port 9020 to receive objects (in the form of JSON messages) containing the header packet information of the network traffic of the other peers belonging to the P2P Network.

4.2 Botnet Detection

The **Botnet Detection module** is the one which starts the analysis of the network traffic. As we saw previously (see subsection 3.3.3), the `TaskQueue` structure is used to create a Queue of "Botnet Detection tasks" to analyze several batches of fetched packets' information passed by the eBPF filter located in Kernel Space (see section 3.3).

Each *Botnet Detection task* performs the following operations:

- The first one is sending the packets information of its own traffic to the other active peers of the P2P Network on port 9020 in form of JSON messages (see section 4.1);
- The second step is performing the Flow-based analysis on the packets informa-

tion of its own network traffic;

- The third step consists in initializing a new *Incremental Learning task* (see section 4.3) by passing, as arguments, the flow features of the packets information of the previously flow-based analyzed "batch", the header packet information on which to perform graph-based analysis and the *flow-based training dataset* to update; then, this task is pushed into the related Incremental Learning `TaskQueue` in order to be executed.

- The last action consists in updating the **suspicious_IPs HASH MAP** (see Figure 4.1) with new IPs predicted as "suspicious" by the flow-based analysis.

4.2.1 Flow-Based analysis

The **Flow-based analysis module** is composed of the following sequential actions:

- Extraction of *flow features* from the header packets information retrieved from the **queue QUEUE MAP** (see Figure 4.1).

- Training the **Random Forest Classifier** (see section 4.4) to build a new Classifier with the updated flow-based training dataset.

- Predicting the *suspiciousness* (the probability of being "suspicious") of the flows (whose features have just been extracted) by using the previously trained **Random Forest Classifier**. Then, it is possible to compute the average of the *suspiciousness* of the several flows involving the same IP of an external host to analyze. If the average *suspiciousness* associated to an external host IP is greater than a specified threshold then that IP is labeled as "suspicious".

Flow Features Extraction

The extraction of flow features is performed by grouping the fetched packets information in "flows" labeled by a "Flow ID" with the following structure: `<IP_1-IP_2-Port_1-Port_2-Protocol>`. In fact, according to [10], "*if two different*

packets have the same source/destination host/port and the same protocol, they belong to the same flow".

Subsequently, every group of packet information belonging to the same flow is passed to a function dealing with the actual extraction of several features. The chosen flow features have been taken from [52, 10, 31, 5, 56] and are shown below.

- **SrcIP** and **DstIp**: Source and destination IP addresses are extensively used in Botnet detection systems and they are needed to quantify the number of distinct connections. Nevertheless, these features do not provide a definitive conclusion, nevertheless they are complementary in the assessment of botnet communication patterns and allow to carry out a simple blacklisting of known IP addresses of C&C controllers [5].

- **SrcPort** and **DstPort**: the source/destination ports are largely employed for detecting botnet traffic, although some types of attacks use a periodic change of source-destination port combination making ineffective these two features (for example, in the case of Nugache botnet) [5]. In other cases, botnets could use ports often used by normal users (such as port 80 for HTTP and port 443) in order to hide their traffic [31].

- **Duration**: this is the total time of the connection from beginning to the end and it is one of the most used feature in botnet detection. This feature becomes very useful for some types of botnets characterized by an unidirectional and short connection followed by much longer communication sessions and other types of botnets known to be 'chatty' whose communication duration may vary (e.g. Palevo botnet, IRC botnets) [31, 5].

- **PX**: the number of the exchanged packets is useful to detect some botnets which send a large number of packets. For example, we can assume that a botmaster needs to send many packets to manage the connections with the bots in order to keep theme alive. So, counting the number of the exchanged packets might be useful to identify this behavior [31, 5].

- **NNP**, **NSP** and **PSP**: these are the number of null packets (i.e. packets with no payload), the number of small packets (i.e packets with length of 63-400

bytes) and the percentage of small packets over the total number of packets, respectively. The small packets are commonly used in decentralized botnets to probe their peers, or in IRC botnets by the bots to exchange small chat messages whit C&C servers. Instead, the usage of null packets has not yet been seen in recent researches [31, 5].

- **IOPR**: the ratio between the number of incoming packets over the number of outgoing packets is a feature whose discriminatory power for detecting a botnet behaviour hasn't been fully proven yet [5]. Some studies as [34], have analyzed backbone-traffic with respect to behaviour differences between inbound and outbound Internet traffic for various protocols; an other research [2] has shown that most of malicious flows over TCP are generally part of incoming traffic.

- **Reconnect**: some types of botnets apply a simple strategy to prevent detection which consists in randomly reconnecting as an established connection. Therefore, this feature can be controlled by counting the number of SYN packets sent by a suspicious host [31].

- **FPS**: the size of first transferred packet in the flow may be useful for detecting specific types of botnets because it reveals some characteristics of the underlying protocol which may identify a malicious behaviour [31, 5].

- The *Flow size features* characterize the length of the flow and are useful to identify similar communication patterns. Typically, the botnet traffic is generated by predefined bot actions and it is more uniform than traffic of legitimate users that, due to variable length messages/commands, is quite irregular. For example, this feature is very effective for detecting the malicious traffic of the Weasel Botnet that uses fixed length commands [5, 21]. Below, some flow size features are listed.

  - **APL**: average payload packet length for time interval;
  - **DPL**: total number of different packet size;
  - **PV**: standard deviation of payload packet length;

  - **TBT**: the total number of transmitted bytes is used to get similarities out of botnet traffic, such as fixed length commands [31].

- As for the *Flow size features*, the average bits-per-second (**BPS**), the number of packets-per-second (**PS**) and the average inter arrival time of the packets (**AIT**), aim to characterize the similarity of network communication [5].

- **MPL**: Maximum of packet length in the flow [56].

- **MP**: Number of maximum packets [56].

Some of the above described features (as **NNP** and **PX**) consider also the case in which the host performing our detection and mitigation method, belongs to a botnet as bot. In this case, our method attempts to detect other bot hosts (belonging to the same botnet) or the botmaster to mitigate the traffic exchanged with them.

4.3 Incremental Learning

As we have seen in section 4.2, the **Incremental Learning module** is invoked by the **Botnet Detection module** with three main arguments: the batch of the fetched header packet information to analyze, the flow-features extracted by the batch, and the flow-based training dataset to update.

Once the Incremental Learning task is started, the header packet information of the active P2P peers received through the Server listening on port 9020 (see section 4.1) are added to the batch of the header packet information passed by the eBPF filter program. In this way, the graph-based analysis is carried out on the network traffic captured by the whole P2P network. After graph-based analysis has been completed, the resulting predictions of the external host IPs are assigned to the features (passed as argument) of the flows involving that IPs and the passed flow-based training dataset is updated. Moreover, like **Botnet Detection Module** (see section 4.2), the **Incremental Learning module** updates the **suspicious_IPs HASH MAP** (see Figure 4.1) with new IPs predicted as "suspicious" by the graph-based analysis.

4.3.1 Graph-Based analysis

The **Graph-Based analysis** starts extracting graph-based features from the retrieved packets information of the traffic of the whole P2P Network. Then, the resulting features are passed to the Random Forest for the classification. The Random Forest Classifier for the Graph-based analysis (unlike the Flow-based analysis) is not retrained for each *header packet information batch* to process, but it is trained only once at the beginning, and then it is shared among all the *Incremental Learning tasks*.

Graph Features Extraction

The **Graph Features Extraction** starts creating a Graph associated to the packets information of the traffic of the whole P2P Network.

Let's assume that the batch of packets information to analyze is a set P containing 7-tuples $p_i = (sip_i, dip_i, ts_i, sp_i, dp_i, ttl_i, nb_i)$. sip_i is the source IP address that uniquely identifies a source host, dip_i is the destination IP address that uniquely identifies a destination host, ts_i is the timestamp of the packet, sp_i is the source port, dp_i is the destination port, ttl_i is the Time to live and nb_i is the number of bytes of the packet. The system creates a graph $G(V, E)$ where V is a set of nodes and E is a set of directed edges $e_{j,i}$ from node v_i. The set of nodes V is defined in the following way [1]:

$$V = \bigcup_{\forall p_x \in P} (\{sip_x\} \cup \{dip_x\}) \tag{4.1}$$

For every $p_x \in P$, there exists directed edges $e_{i,j}$ and $e_{j,i}$ from v_i to v_j and v_j to v_i, respectively, such that $sip_x = v_i$ and $dip_x = v_j$. Therefore:

$$E = \bigcup_{\forall p_x \in P} (\{(sip_i, dip_i, ts_i, sp_i, dp_i, ttl_i, nb_i)\}) \tag{4.2}$$

After having creating the Graph G, the *Graph features Extraction* function computes twelve Graph features from G.

- **In-degree (ID)** and **Out-degree (OD)**: The in-degree, $ID(v)$, and out-degree, $OD(v)$, of a node $v \in V$ are the number of its ingress and egress edges,

respectively.

$$\mathcal{F}(e_{i,j}) = \begin{cases} 1, & \text{if } e_{i,j} \in E \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

$$ID(v_i) = \sum_{v_j \in V,\ v_i \neq v_j} \mathcal{F}(e_{j,i}) \tag{4.4}$$

$$OD(v_i) = \sum_{v_j \in V,\ v_i \neq v_j} \mathcal{F}(e_{i,j}) \tag{4.5}$$

As said previously in section 4.2.1, the malicious hosts may be detected examining the amount of incoming and outgoing network traffic. Typically, malicious hosts with a higher ID might mean an attempt to infect some network; however, a high ID may not signify malicious activity as in the case of a gateway, that being a central point of communication in a network, it has a high ID but it is not necessarily a malicious endpoint. Instead, a high OD occurs during the *reconnaissance stage* of the *intrusion kill-chain* where bots select the targets, researches them and attempt to identify vulnerabilities in the target network [1].

- **Number of in-neighbors**: number of the adjacent vertices of a vertex v which have an edge to v.

- **Number of out-neighbors**: number of the adjacent vertices of a vertex v which have an edge from v.

- **Page Rank (PR)**: it is an algorithm that measures the "importance" of the nodes in a graph by assigning to each node a rank. The value of PageRank of vertex v, $PR(v)$, is given iteratively by the relation:

$$PR(v) = \frac{1-d}{N} + d \sum_{u \in \Gamma^-(v)} \frac{PR(u)}{d^+(u)}$$

where $\Gamma^-(v)$ are the in-neighbors of v, $d^+(u)$ is the out-degree of u, and d is a damping factor [24].

- **Betweenness Centrality (C_B)**: the betweenness centrality of a node $v \in V$, is a measure of the number of shortest paths that pass through it. It is defined

as

$$C_B(v) = \sum_{s \neq v \neq t \in V \;\; s \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4.6}$$

where σ_{st} is the number of shortest paths from s to t, and is the number of shortest paths from s to t that pass through a vertex v. The algorithm to compute this feature has a complexity of $\mathcal{O}(VE)$ for unweighted graphs and $\mathcal{O}(VE + V(V + E) \log V)$ for weighted graphs. The space complexity is $\mathcal{O}(VE)$ [24]. This feature, as explained in [1], *"can alienate bots early on as they attempt their first connections. This is when the bots exhibit low IDW and ODW. Thus, it would be more favorable for the shortest paths in the network to pass through the host. Likewise, when the IDW and ODW increase, the BC of a node decreases immensely, as it is less favored for being included in shortest paths"*.

- **Closeness Centrality (C)**: in a connected graph, the closeness centrality of a node v is defined as the reciprocal of the sum of the length of the shortest paths between the node v and all other nodes in the connected graph. So,

$$C(v_i) = \frac{1}{\sum_{v_j} d(v_j, v_i)} \tag{4.7}$$

where $d(v_j, v_i)$ is the *distance*[1] between vertices v_i and v_j. If there is no path between the two vertices, the distance is considered to be zero [24].

- **Eigenvector Centrality (EC)**: the eigenvector centrality c_i of a node v_i is defined as the weighted sum of the centralities of all nodes that are connected to it by an edge, $A_{i,j}$. So,

$$c_i = \epsilon^{-1} \sum_{j=1}^{n} A_{i,j} c_j \tag{4.8}$$

where c is the eigenvector associated to the eigenvalue ϵ of A (the adjacency matrix with elements A_{ij} defining the strength of the physical correlation between nodes i and j). The EC is a measure of the number and the quality of the connections of a node with other well-connected nodes in the graph: a

[1] In graph theory, the distance between two vertices in a graph is the number of edges in a shortest path connecting them

vertex with a smaller number of high-quality contacts may outrank one with a larger number of mediocre contacts [44, 24].

- **Katz Centrality**: Katz centrality is the measure of topological centrality that helps to discover the relative influence of each node on the graph by taking into account it's immediate neighboring nodes as well as non-immediate neighboring nodes that are connected through immediate neighboring nodes. Te Katz centrality of a node v_i is computed as:

$$C_{Katz}(v_i) = \alpha \sum_{j=1}^{n} A_{j,i} C_{Katz}(v_j) + \beta \tag{4.9}$$

where α is a constant called the damping factor, usually considered to be less than the largest eigenvalue, γ i.e. $\alpha < 1/\gamma$ and β is a bias constant, also called the exogenous vector, used to avoid the zero centrality values. With $\alpha \geq \gamma$, the centrality tends to diverge [59].

- **Authority Centrality** and **Hub Centrality**: these two scores are computed through the Hyperlink-Induced Topic Search (HITS) algorithm that assigns hub and authority centralities to the vertices. Assuming $h(v)$ and $a(v)$ denoting the hub and the authority score, we set, initially, $h(v) = a(v) = 1$ for all nodes v. We also denote by $v \mapsto y$ the existence of an edge from v to y. The core of the iterative algorithm is a pair of updates to the hub and authority scores of all nodes given by Equation 4.10 and Equation 4.11, which capture the intuitive notions that good hubs point to good authorities and that good authorities are pointed to by good hub

$$h(v) \leftarrow \sum_{v \mapsto y} a(y) \tag{4.10}$$

$$a(v) \leftarrow \sum_{y \mapsto v} h(y) \tag{4.11}$$

Thus, the first line of Equation 4.10 sets the hub score of node v to the sum of the authority scores of the nodes it links to. If v links to pages with high authority scores, its hub score increases. The Equation 4.10 plays the reverse role: if node v is linked to by good hubs, its authority score increases [43].

- **Local Clustering coefficient (LCC)**: this features is characterized by a computational overhead lower than C_B and it quantifies the neighborhood connectivity LCC_i of a node $v_i \in V$, such that

$$LCC_i = \frac{\sum_{v_j, v_k \in N_i,\ v_i \neq v_j \neq v_k} \mathcal{F}(e_{j,k})}{|N_i|(|N_i| - 1)} \tag{4.12}$$

  where N_i is the neighborhood set for v_i, $\forall v_j \in N_i \mid e_{i,j} \in E \vee e_{j,i} \in E$. Typically, the hosts which have been successfully infected, exhibit a higher LCC [1].

Other features (similar to flow features but per node instead of per flow) are computed relying on the tuples $p_i \in P$. They are: *Average incoming packet size*, *Max incoming packet size*, *Min incoming packet size*, *Average outgoing packet size*, *Max outgoing packet size*, *Min outgoing packet size*, *Number incoming bytes*, *Number outgoing bytes*, *Number source ports*, *Number destination ports*, *Average incoming TTL*, *Max incoming TTL*, *Min incoming TTL*, *Average outgoing TTL*, *Max outgoing TTL*, *Min outgoing TTL*.

4.4 Random Forest Classifier

As we have seen in subsection 4.2.1 and in subsection 4.3.1, the classification of network traffic is carried out through the Random Forest.

Unlike standard trees, the Random Forests add an additional layer of randomness to bagging. In addition, each node is no longer split using the best split among all variables but among a subset of predictors randomly chosen at that node. This method is very powerful with respect to others type of classifiers like support vector machines and neural networks, and it is characterized by a good robustness to overfitting. The Random forests run relying on two parameters: the number of variables in the random subset at each node and the number of trees in the forest [38].

4.4.1 Random Forest algorithm

The algorithm of the Random Forest follows the steps below [38, 8].

1. Draw n_{tree} bootstrap samples at random (but with *replacement*) from the original data to create a subset of the total set.

2. For each of the bootstrap samples, create a tree: randomly choose a subset m of variables and then, at each node, find in m a variable (and a value for that variable) which optimizes the split. Each tree is grown to the largest extent possible without applying any pruning.

3. Each created tree outputs a prediction. The Forest aggregates the predictions of the n_{tree} trees and chooses the classification having the most votes.

The Random Forest error rate depends on two factors [8]:

- The *correlation* between any two trees in the forest. The correlation is directly proportional to the forest error rate.

- The *strength* of each individual tree in the forest. The strength is inversely proportional to the forest error rate.

Moreover, m is directly proportional to *correlation* and the *strength*. The error rate can be obtained relying on the training data, following these two steps [38]:

1. At each bootstrap iteration, predict the data not in the bootstrap sample ("out-of-bag", or OOB data) using the tree previously created with the bootstrap sample.

2. Compute the OOB estimate of errorate by aggregating the previously obtained OOB predictions.

In [8], Leo Breiman introduces two properties of the Random Forests: the *strength* and the *correlation*. These two properties are defined through an upper bound of the generalization error noted $PE*$

$$PE^* \leq \frac{\bar{\rho}(1-s^2)}{s^2} \tag{4.13}$$

where s is the *strength* and ρ is the *correlation*. From Equation 4.13, it can be said that the lower the ratio $\frac{\bar{\rho}}{s^2}$, the more chances to obtain a low error rate (i.e. a better forest).

The margin function of a RF is defined by the following equation:

$$mr(x, y) = P_\Theta(h(x, \Theta) = y) - \max_{j \neq y} P_\Theta(h(x, \Theta) = j) \tag{4.14}$$

where x is an input data, y its class, and where the subscripts Θ indicate that the probability is over the Θ_k family of random vectors. The *strength* is then defined as the expectation of this margin over the data space:

$$s = E_{x,y}[mr(x, y)] \tag{4.15}$$

The raw margin function is defined as:

$$rm(\Theta, x, y) = I(h(x, \Theta) = \hat{j}(x, y)) \tag{4.16}$$

where $\hat{j}(x, y)$ is the index of the "best" class among the wrong classes, and it is defined as:

$$\hat{j}(x, y) = \arg\max_{j \neq y} P_\Theta(h(x, \Theta) = j) \tag{4.17}$$

The *correlation* is actually the statistical mean correlation between $rm(\Theta, x, y)$ and $rm(\Theta', x, y)$ over all pairs of (Θ, Θ') [4].

4.5 Mitigation

The mitigation approach used in the proposed methodology, as we already saw in section 4.2 and section 4.3, relies on the predictions passed from userspace to kernel space through the **suspicious_IPs HASH MAP** (see Figure 4.1): a map containing a set of IPs of external hosts which are considered "suspicious" after the userspace botnet analysis performed on the *header packet information* (contained in a `struct Packet` structure in Listing 3.5) received by the eBPF Program.

The policy adopted by the proposed methodology to update the map with new detected *suspicious IPs* is the following:

- If the host executing the Botnet detection system is connected with other peers in the P2P network (see section 4.1), the graph-based predictions have more priority than flow-based ones. Different priorities are assigned to the outcomes of the two types of analysis since the flow-based detection is more

responsive than graph-based one, but less accurate, so the graph-based analysis can remove from the map the false malicious IPs resulting by the flow-based analysis.

- If the host singularly runs the Botnet Detection system, it is necessary relying only on flow-based analysis since the graph-based one, as already explained in section 4.3, loses accuracy for lack of botnet network traffic got from other peers of the P2P network. Moreover, the Incremental Learning allows to enhance the accuracy of the flow-based analysis in order to perform quite well during the future flow-based analysis executions on botnet traffic patterns already analyzed.

Naturally, this "map update policy" is no longer effective if we have that the graph-based analysis is less accurate than the flow-based analysis as a result of an insufficient number of active peers in the P2P network that prevents the graph-based analysis to be carried out properly. In fact, in this situation, the graph built by a host with a very limited botnet network traffic will most likely not be able to identify malicious hosts.

So, the eBPF program reads the IPs listed in the **suspicious_IPs HASH MAP** and rejects all incoming/outgoing packets from/to those IPs (as shown in Listing 4.1 at lines 28,32).

All the packets which are not present in **suspicious_IPs HASH MAP** and all the packets coming from the IPs present in the **P2P_IPs HASH MAP** (see Figure 4.1) and from ports involved by the **P2P Module** (see Figure 4.1), are allowed through by returning the value `-1` as we can see at lines 10,17,36 in Listing 4.1.

```
1 // ...
2 int ebpf_program(struct __sk_buff *skb) {
3
4     // ...
5
6   if ((packet.protocol == 6) || (packet.protocol == 17)){
7     if (P2P_IPs.lookup(&packet.dst_ip) != NULL){
8       if ((packet.src_port == 9020) || (packet.dst_port == 9020) || (
    packet.src_port == 8000) || (packet.dst_port == 8000)){
```

```
9         bpf_trace_printk("ALLOWED packet: 'dst_ip' in 'P2P_IPs' hash
    and '(dst/src)_port' used in P2P network\n");
10        return -1;
11      }
12    }
13    else{
14      if (P2P_IPs.lookup(&packet.src_ip) != NULL){
15        if ((packet.src_port == 9020) || (packet.dst_port == 9020) ||
     (packet.src_port == 8000) || (packet.dst_port == 8000)){
16          bpf_trace_printk("ALLOWED packet: 'src_ip' in 'P2P_IPs'
    hash and '(dst/src)_port' used in P2P network\n");
17          return -1;
18        }
19      }
20    }
21
22    queue.push(&packet, BPF_EXIST);
23  }
24
25
26  if (packet.src_ip == localIP && packet.dst_ip != localIP &&
    suspicious_IPs.lookup(&packet.dst_ip) != NULL){
27    bpf_trace_printk("BLOCKED packet: 'dst_ip' in 'suspicious_IPs'
    hash\n");
28    return 0;
29  }
30  else if (packet.src_ip != localIP && packet.dst_ip == localIP &&
    suspicious_IPs.lookup(&packet.src_ip) != NULL){
31    bpf_trace_printk("BLOCKED packet: 'src_ip' in 'suspicious_IPs'
    hash\n");
32    return 0;
33  }
34  else{
35    bpf_trace_printk("ALLOWED packet\n");
36    return -1;
37  }
38}
```

Listing 4.1. eBPF program mitigation

Chapter 5

Experimental Assessment

In this chapter, we present a large number of experimental results obtained by relying on traces obtained from real-world botnets. The goal of this experimental study is twofold. On the one hand, we want to stress test the proposed approach, and see whether it is capable of providing good detection results also *while* a botnet-based attack is being carried out—we recall that this is the enabling factor to perform an effective mitigation of the attack. On the other hand, we want to experimentally assess what are the requirements from the distributed system to make our approach viable and effective, and what could be possible lines of improvement to introduce in future work.

5.1 Tested Botnets

The dataset used to train the Random Forest Classifiers of the Flow and Graph based analysis has been taken from the University of New Brunswick [7]. From the *UNB pcap training dataset*, the effective training datasets (for Flow and Graph based detection) have been extracted: they have been created extracting the flow/graph features from UNB pcap training dataset and storing in hdf5 files[1].

In addition to the *UNB pcap training dataset*, the University of New Brunswick provides also the *UNB pcap testing dataset* to test the botnet detection methodologies.

[1] In the case of the flow-based detection, the dataset is represented in Figure 4.1 as "Flow-based Training dataset".

The first dataset, which has been used to create our "Flow-based Training dataset" and "Flow-based Training dataset" files, contains samples from the following botnet:

- **Neris**: a botnet that uses an IRC C&C channel to communicate with its bots, attempting to send SPAM for performing click-fraud using some advertisement services [42].

- **Rbot**: an old-school IRC botnet that uses the "Rbot malware kit". It has not the same scalability of other well-known botnets. The Rbot's underlying malware uses a backdoor to gain control of the infected machine (installing keyloggers, viruses, and even stealing files from the machine) for sending spam or performing DDOS attacks [30].

- **Virut**: a botnet which spreads by injecting code into any executable or screensaver file that is accessed. It also injects malicious iframes into HTML, PHP and ASP files. The distribution of this malware to additional devices can be carried out by network shares, removable drives and "malvertising"[2]. Moreover, it uses a domain generation algorithm and an encrypted protocol with RSA signature verification for C&C signalling. After Virut establishes the connection with victims machines, it instructs them to download a portable executable file. This attempts to drop further payloads over HTTP, using the user agent 'AdInstall' [45].

- **NSIS**: a type of Trojan behaving as a P2P Botnet that targets Windows platform. It instructs the infected machines to download and install additional malware [20].

- **SMTP Spam**: it refers to some types of botnets which spoof a return address and easily mail the same message to multiple recipients [14].

- **Zeus**: a financial malware which waits for infected machines to log into a list of targeted banks and financial institutions, and then steals their credentials by recording the keystrokes used to log in. Then, the malware sends these

[2]Malvertising is the insertion of malicious advertisements into otherwise legitimate webpages or advertising networks.

information to a remote C&C server in real time. Zeus can appear in two botnet forms: centralized and decentralized (P2P) (see section 2.1). In the latter case, this botnet is already known as "GameOver Zeus" which, unlike centralized Zeus version, has a peer-to-peer C&C infrastructure meaning that the instructions to the infected computers can come from any of the infected computers, making a takedown of the botnet more difficult [17, 37, 32].

From the second dataset, we have extracted a pcap file (which is a sub-selection of the *UNB pcap testing dataset*) for each considered botnet. Each of these pcap files contains the traffic of some normal hosts which communicate with a malicious host infected by a specific botnets. The botnet considered for the test are divided in two classes: *known* and *unknown*. The first class, unlike the second one, is composed by samples of botnets present also in the "UNB training dataset" (which were described above). The tested *known* botnets are: Neris, Virut and Zeus. The *unknown* botnets are used to test the generalizability and the botnet-agnosticism of our botnet detection methodology. The considered *unknown* botnets are:

- **Menti**: an IRC-based botnet. The main activities are to employ a custom unencrypted protocol to connect to C&C server and to scan SMTP servers (i.e., TCP port 25) [47].
- **BlackHole**: a P2P Botnet.
- **Weasel**: a botnet designed for providing an open-source platform for analysis of botnet traffic. It is characterized by a realistic and secure communication channel allowing a secure communication between bots and the botmaster. The Weasel framework is composed of distinct modules. The bot module can be viewed as a server which listens for socket connections from the C&C module. The C&C subsystem is invoked on demand as the botmaster wishes to send commands. Both the C&C and bot modules interact with a RESTful Python web service, which contains functionality for updating bot status and maintaining a history of sent commands. Weasel makes use of a shared PostgreSQL database directly accessible from C&C modules, whereas the bot has indirect access through the web service. Weasel provides an HTTPS-

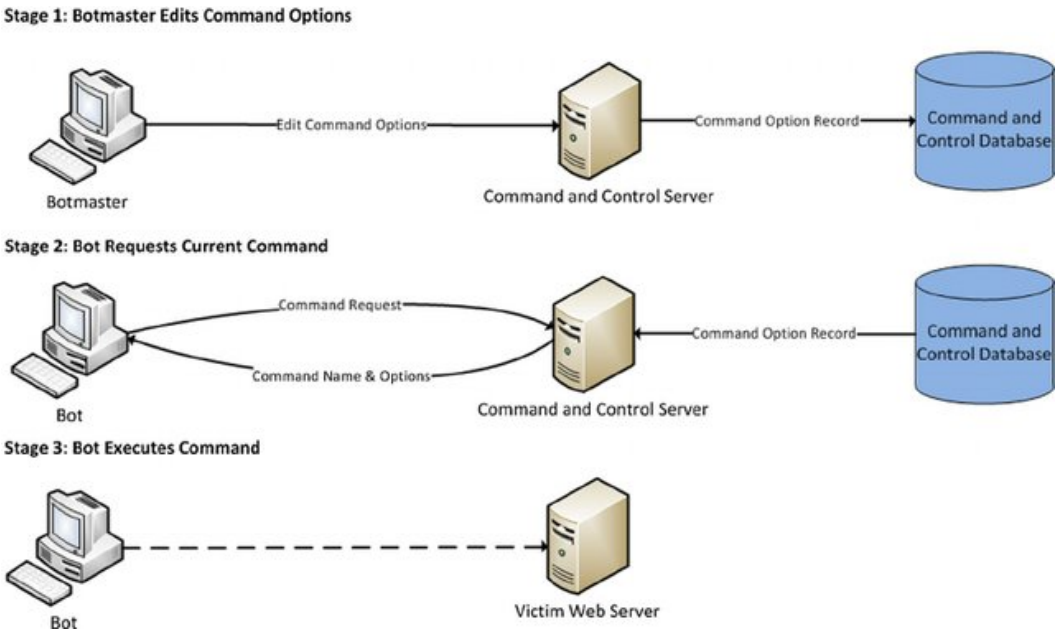


Figure 5.1. Weasel communication overview [60].

encrypted web interface (that gives a read-only access to bot status information, command history, and user accounts) whose access is controlled with the C&C database account table, on the basis of client IP address and a password such that information being transmitted within the bot network is visible to only those hosts which the botmaster designates. When a bot starts up, it executes a command from the botmaster, or shuts down, it issues an "activity notification" (in form of HTTP GET request) to the C&C server such that the botmaster knows if the bot is active or not. Moreover, to prevent hijacking of the botnet, bots perform basic message authentication & validation: upon receipt of a command from the botmaster, the bot sends a validation request to the C&C server, containing the MD5 hash of the received command. The server scans the history of commands issued within the past 10 seconds, and if there is a match between the stored hash value and the hash value received by the bot, the C&C server indicates that the command is valid by sending to the bot a "validation indicator". The Weasel botnet can be used for DDoS attacks, spamming, or even distributed computing. The communication stages performed by the Weasel Botnet are shown in Figure 5.1 [60].

5.2 Tests description

The tests performed are three:

- **Single-host Test**: this test is performed on a host without interacting with other P2P hosts. In this way, it is possible to see the behaviour of our approach running on a non-distributed system.
- **P2P Test**: this test is performed by running the tool on all P2P hosts at same time. In this way, it is possible to see the behaviour of our approach running on a distributed system.
- **Incremental Learning Test**: this test is performed after doing the *P2P Test* by enabling only the flow-based analysis. In this way, it is to possible to see the effects of the Incremental Learning of the P2P Test on the Flow based analysis of this test.

The P2P tests will be compared to an "Oracle" which performs an graph-based botnet detection all at once of the whole traffic exchanged by all peers of the P2P network throughout the duration of the tests. The "Oracle", despite having an high accuracy and a low time of execution, has no way of performing an online detection and mitigation, as opposed to our approach.

5.3 Experiment

The results shown below have been obtained by developing the *AntiBotnet* tool[3]: an implementation of our approach. The implementation has been tested on Amazon Web Services EC2 *t3.medium* instances which allow to emulate a distributed environment on a local network. The *t3.medium* instances have the following features:

- 1st or 2nd generation Intel Xeon Platinum 8000 series processor (Skylake-SP or Cascade Lake) with a sustained all core Turbo CPU clock speed of up to 3.1 GHz. Additionally there is support for the new Intel Advanced Vector Extensions 512 (AVX-512) instruction set, offering up to 2x the FLOPS per core compared to the previous generation T2 instances.

[3]The source code is available at `https://github.com/alessioizzillo/AntiBotnet`.

- 2 vCPUs (i.e. threads of a CPU core).
- Memory size of 4 GB.
- Network burst bandwidth of 5 Gbps.
- EBS burst bandwidth of 5 Mbps

The number of instances used for testing the implementation on each botnet, are: 5 for Neris, 5 for Virut, 5 for Zeus, 2 for Menti, 5 for Weasel and 4 for BlackHole.

The test pcap files extracted by the *UNB pcap training dataset* (see section 5.1) are replayed on the host on which the tool (implementing our approach) runs in order to emulate the attack of the botnets on these hosts. The replay of the test pcap files related to some botnet have been carried out by replacing the IP of the hosts which communicate with the malicious host, with the IP of the actual host on which the tool runs. Moreover, in the *Incremental learning Test*, the original IP of the bot hosts has been replaced with a fake IP. In this way, the bots which will communicate with the hosts on which the tool runs, will appear as another host in order to avoid a trivial botnet detection that succeeds only because the bot IP matched exactly the one present in the flow-based training dataset (updated by the graph-based detection in the previously executed *P2P Test*).

5.3.1 Tool usage details

The AntiBotnet tool can be launched through the "AntiBotnet.py" file by typing the following command:

```
sudo python3 AntiBotnet.py mode interface n_pkts n_rf_est_fbd
n_rf_est_gbd test_pcap test_victim_IPs_file pos_victim_IP2replace
test_malicious_IPs_file P2P_IP
```

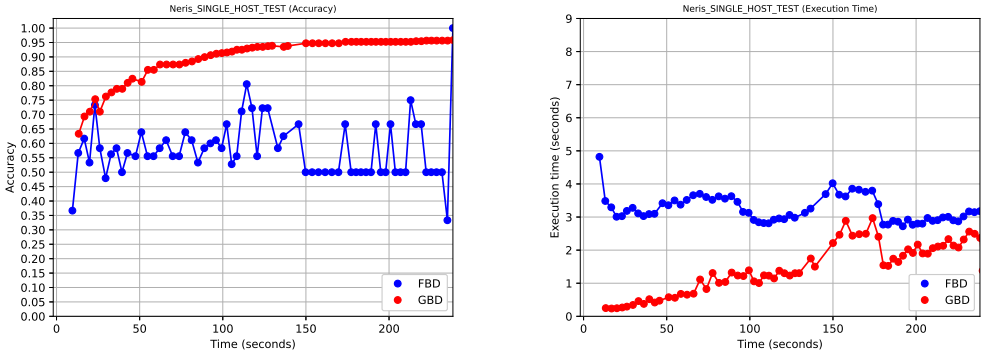
Below, there is the description of the arguments passed to the script through the command above.

- **mode** (mandatory for both modes): the execution mode, "real-world mode" or "test mode";

- **interface** (mandatory for both modes): the Ethernet interface name from which to capture the traffic to analyze;
- **n_pkts** (mandatory for both modes): the size of the batch of packets to analyze;
- **n_rf_est_fbd** (mandatory for both modes): the number of estimators to train the Random Forest Classifier in Flow-Based detection;
- **n_rf_est_gbd** (mandatory for both modes): the number of estimators to train the Random Forest Classifier in Graph-Based detection;
- **test_pcap** (only for "test mode"): path of pcap file from which to replay the packets for testing the tool;
- **test_victim_IPs_file** (only for "test mode"): path of the text file where all the victim IPs which communicate with the malicious host are listed (one per line);
- **pos_victim_IP2replace** (only for "test mode"): position (in the the file "test_victim_IPs_file") of the victim IP to assign to the host where the tool runs;
- **test_malicious_IPs_file** (only for "test mode"): path of the testfile where malicious IPs of test pcap file are listed (one per line);
- **P2P_IP** (optional for both modes): IP of an active host of the P2P network to connect to.

5.3.2 Results on known botnets

Considering the Neris botnet, we can see that in the *Single-host Test*, graph-based detection (gbd) accuracy (Figure 5.2a) grows over the time and, after 150 seconds of execution, it settles at 0.95. Instead, in the case of the flow-based detection (fbd), the accuracy varies from 0.50 to 1.0. The *P2P Test* was performed with 5 peers communicating with the malicious host. Comparing its results (Figure 5.4a) with those in *Single-host Test*, we can notice that the accuracy of the graph-based



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.2. Neris Botnet, Single-host Test.

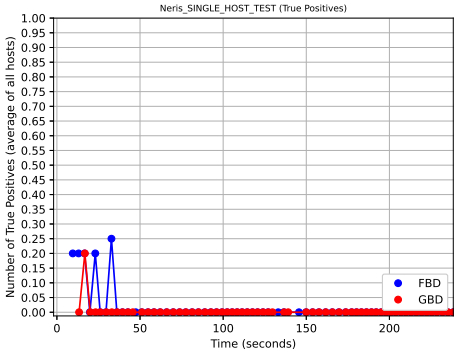
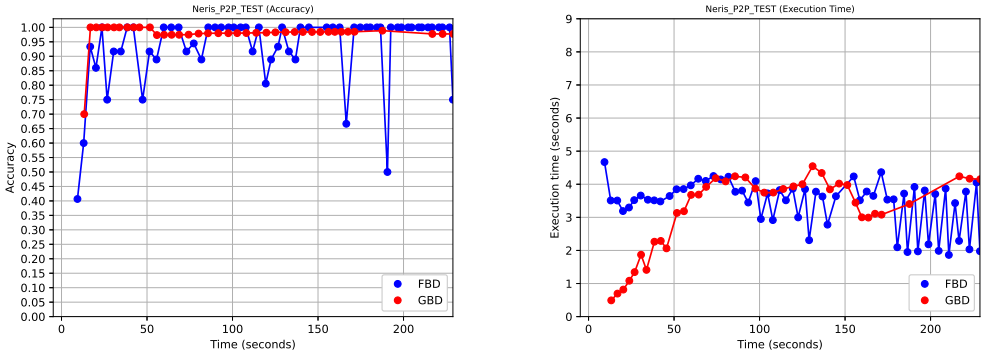


Figure 5.3. Neris Botnet, Single-host Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.4. Neris Botnet, P2P Test.

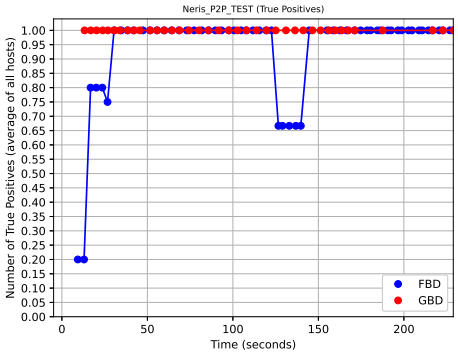
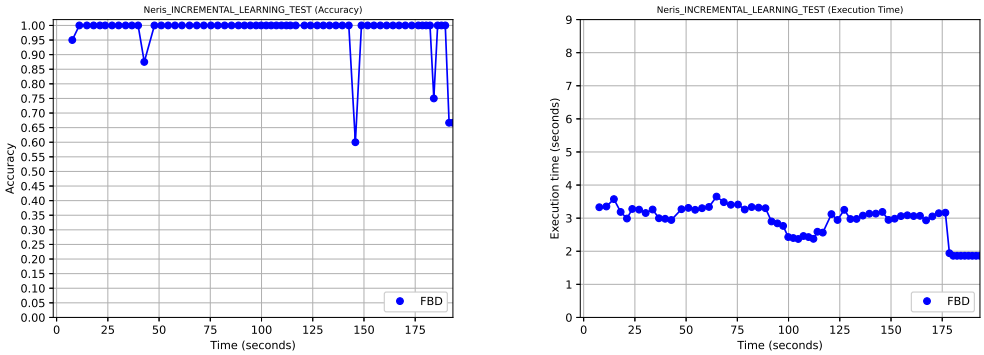


Figure 5.5. Neris, P2P Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.6. Neris Botnet, Incremental Learning Test.

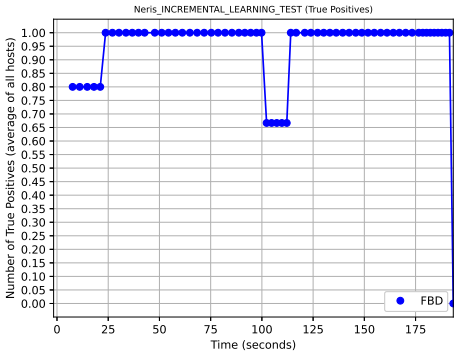
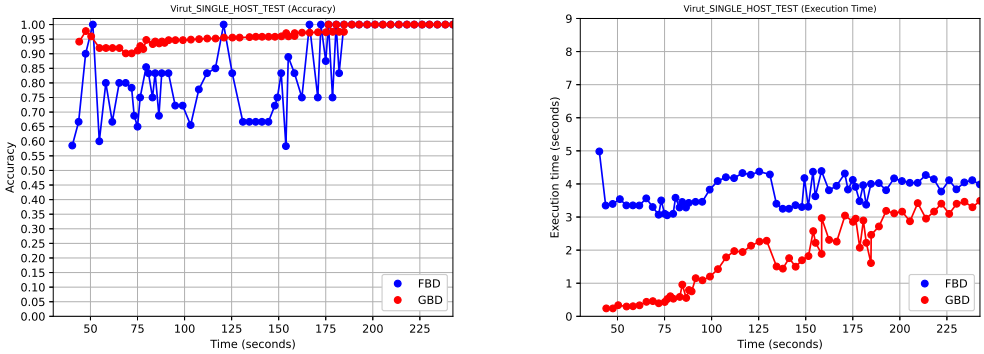


Figure 5.7. Neris, Incremental Learning Test, True Positives.

detection is near to 1 since $Time = 20s$ and, due to incremental learning, the accuracy of the flow-based detection is higher. This means that the host's knowledge

of the network traffic exchanged among all peers of the P2P Network communicating with the malicious host, as expected, gives better results in detecting the malicious host. A further evidence of the incremental learning effectiveness is shown comparing the Flow-based detection True Positives in the *Single-host Test* (Figure 5.3) with those in *Incremental Learning Test* (Figure 5.7): in fact, the incremental learning improves the accuracy of flow-based analysis on the samples of botnets already analyzed in the past by updating everytime the flow-based training dataset. We can perform another evaluation of our method comparing the results of the *P2P Test* (Figure 5.4 and Figure 5.5) with the "Oracle" results. The accuracy of the Oracle is nearly 0.991, the True Positives has a value of 1.0 and the Oracle detection is performed in nearly 2.141 seconds. We can clearly state that the Oracle is more accurate and responsive than our method in "P2P mode" (i.e. when it performs the analysis communicating with others peers of the P2P Network) if we consider a botnet the host has never seen before. Instead, as shown in *Incremental Learning Test* results (Figure 5.6a and Figure 5.7), the analysis on samples of previously analyzed botnets, causes an increase of flow-based method accuracy that overcomes, due to the incremental learning, the graph-based accuracy of the Oracle. The execution times of the graph-based analyses in the *P2P test* (Figure 5.4b) are slightly lower than those of flow-based analyses in *Incremental Learning Test* (Figure 5.6b) making the general approach slightly more responsive when analyzes samples of already analyzed botnets.

Also for Virut, the *P2P Test* was performed with 5 peers communicating with the malicious host. The difference between the accuracy of the *Single-host Test* and *P2P Test* (Figure 5.8a and Figure 5.10a) is negligible both for graph-based and flow-based detection, but the True Positives of the graph-based detection in *P2P Test* (Figure 5.11) are considerably higher than those of the graph-based detection in *Single-host Test* (Figure 5.9). This means, as said for Neris, that the host's knowledge of the network traffic exchanged by the all peers of the P2P Network communicating with the malicious host, improves the capability to detect the malicious host. As in the Neris case, also in Virut incremental learning improves the accuracy of flow-based analysis on the samples of botnets already analyzed in the past by updating everytime



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.8. Virut Botnet, Single-host Test.

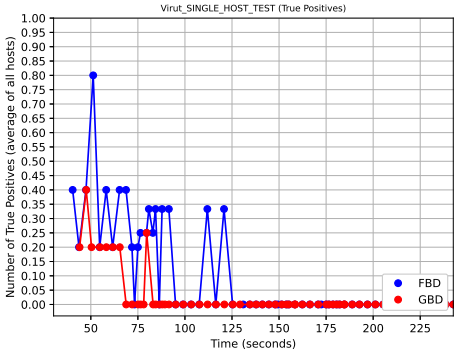
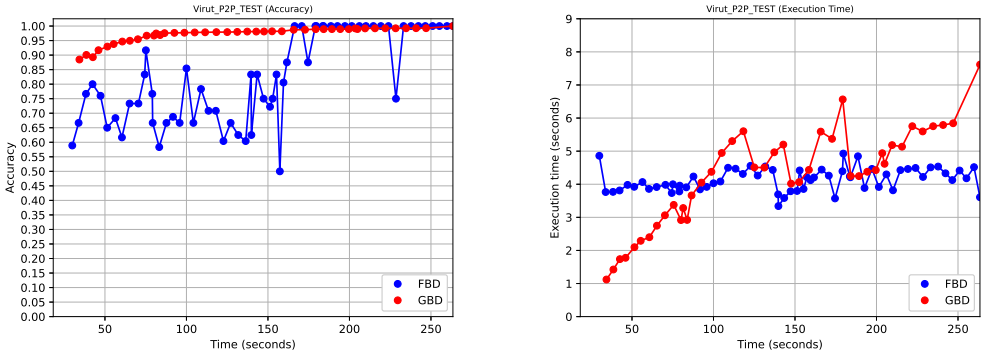


Figure 5.9. Virut Botnet, Single-host Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.10. Virut Botnet, P2P Test.

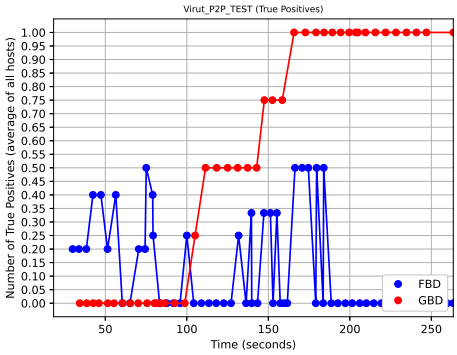
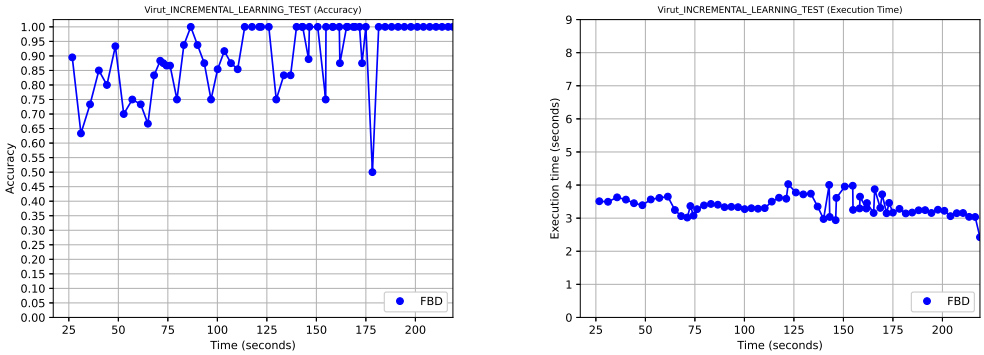


Figure 5.11. Virut, P2P Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.12. Virut Botnet, Incremental Learning Test.

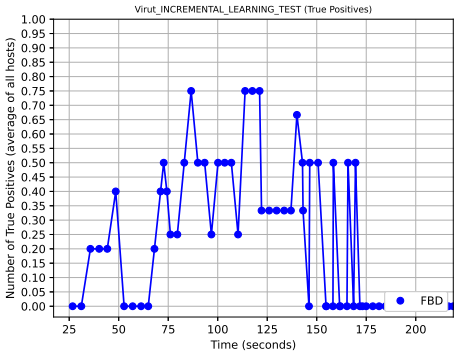
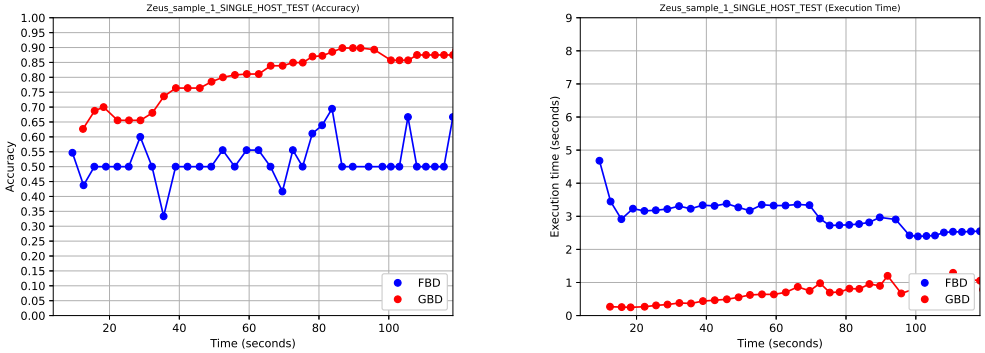


Figure 5.13. Virut, Incremental Learning Test, True Positives.

the flow-based training dataset, as we can see by comparing the True Positives of the *Single-Host Test* (Figure 5.9) and those of the *P2P Test* (Figure 5.13). The

Oracle of the Virut Botnet has an accuracy of 0.987, a True Positives value of 0.8 and an execution time of 3.189 seconds. This accuracy, unlike Neris, is not fully achieved on samples of botnet already analyzed by the flow-based detection of the *Incremental Learning Test* (Figure 5.12a), but it oscillates approximately between 0.50 and 1.0 and this irregularity can be seen also in the True Positives results in Figure 5.13. So, in this case incremental learning is not very effective.



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.14. Zeus Botnet, Single-host Test.

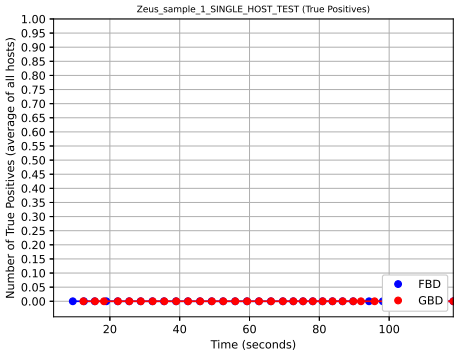
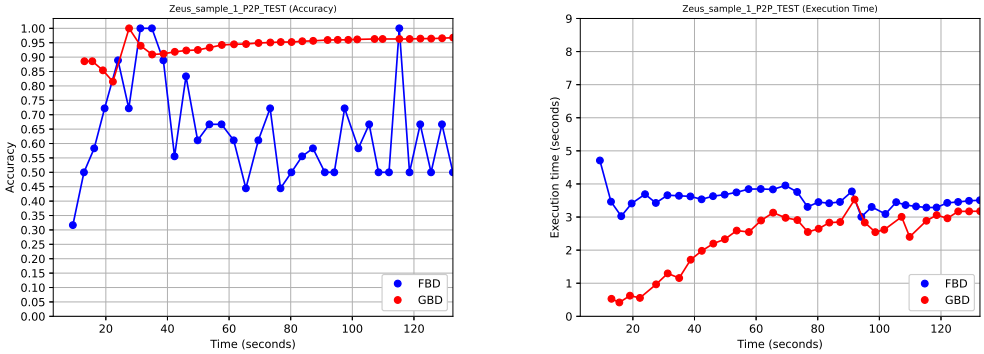


Figure 5.15. Zeus Botnet, Single-host Test, True Positives

Considering the Zeus botnet, we can see that in the *Single-host Test*, graph-based detection (gbd) accuracy (Figure 5.14a), as in the case of Neris, slightly grows over time and, approximately 80 seconds into the experiment, it settles between 0.85 and 0.90. Instead, in the case of the flow-based detection (fbd), the accuracy varies from 0.33 to 0.70. The *P2P Test* was performed with 5 peers communicating with the



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.16. Zeus Botnet, P2P Test.

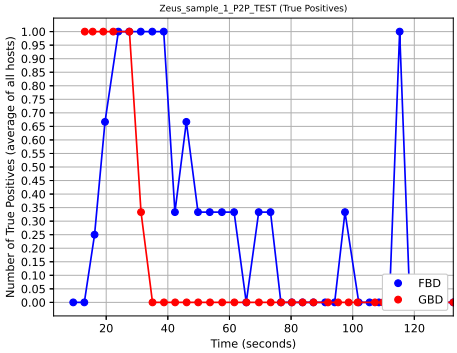
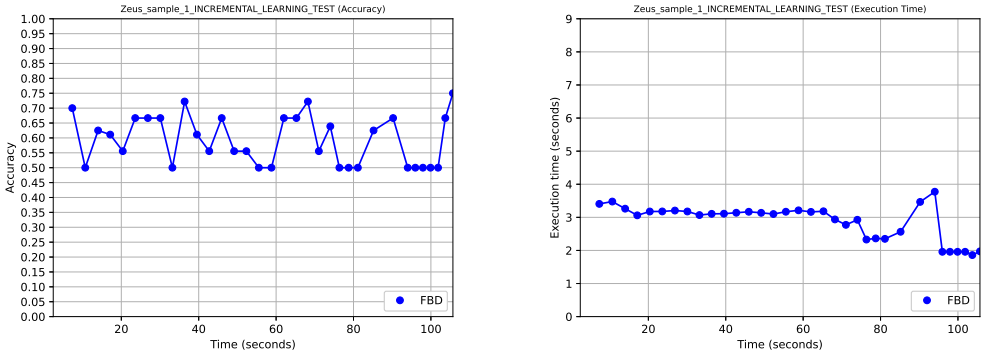


Figure 5.17. Zeus, P2P Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.18. Zeus Botnet, Incremental Learning Test.

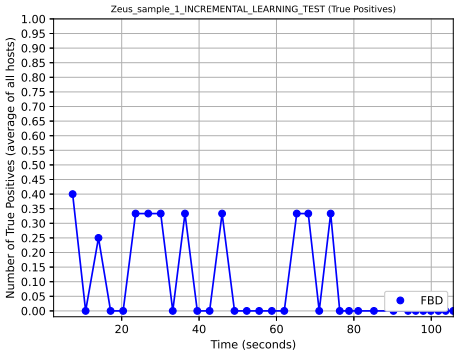
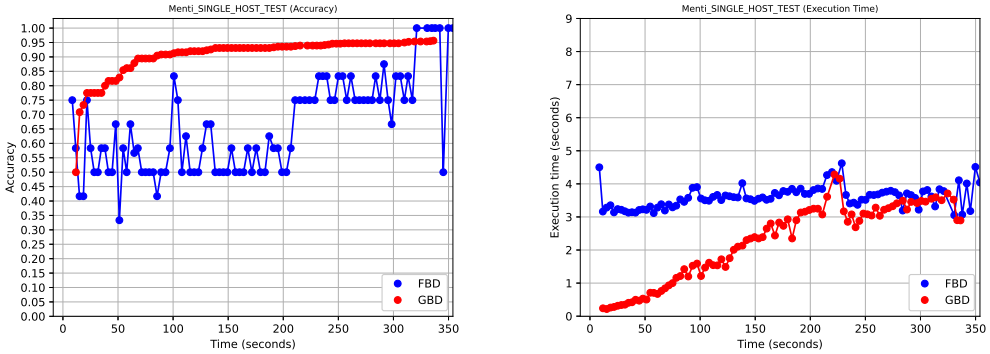


Figure 5.19. Zeus, Incremental Learning Test, True Positives.

malicious host and comparing its results (Figure 5.16a) with those in the *Single-host Test*, we can notice that the accuracy of the graph-based detection is near 0.95 since $Time = 60s$ and, due to incremental learning, the accuracy of the flow-based detection is generally higher due to some peaks achieving more than 0.85 at some time. A further evidence of the incremental learning effectiveness is shown comparing the Flow-based detection True Positives in the *Single-host Test* (Figure 5.15) with those in *Incremental Learning Test* (Figure 5.19): in the first test, the malicious host is not detected at any time, whereas, in the other test, we can notice that the incremental learning has improved the accuracy of flow-based analysis on the samples of botnets already analyzed in the past by updating everytime the flow-based training dataset. We can perform another evaluation of our method comparing the results of the *P2P Test* (Figure 5.16 and Figure 5.17) with the "Oracle" results. The accuracy of the Oracle is nearly 0.956, the True Positives has a value of 0.4 and the Oracle detection is performed in nearly 1.341 seconds. Unlike Neris, as shown in *Incremental Learning Test* results (Figure 5.6a and Figure 5.7), the analysis on samples of previously analyzed botnets, do not cause any significant increase of flow-based method accuracy. Nevertheless, there is a small improvement introduced by Incremental Learning in the True Positives results of the flow-based detection as shown comparing Figure 5.15 with Figure 5.19.



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.20. Menti Botnet, Single-host Test.

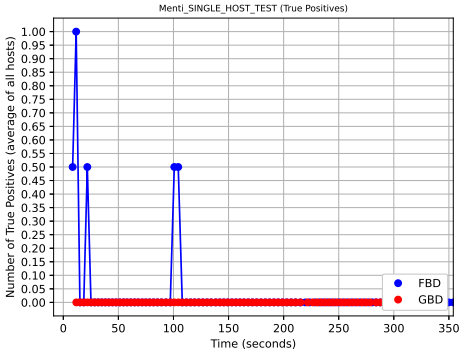
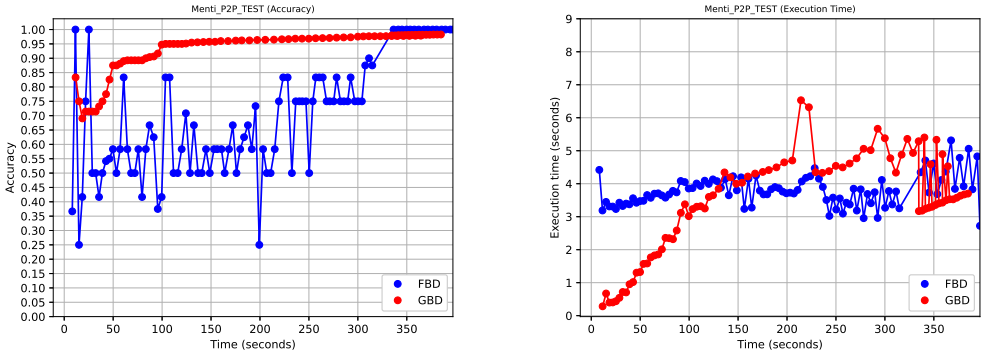


Figure 5.21. Menti Botnet, Single-host Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.22. Menti Botnet, P2P Test.

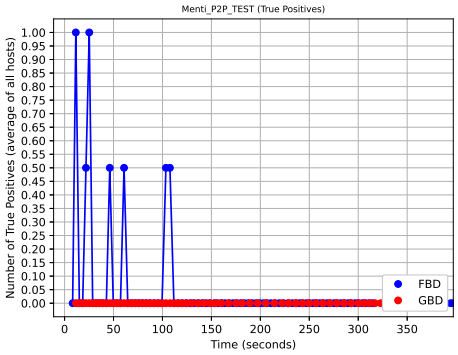
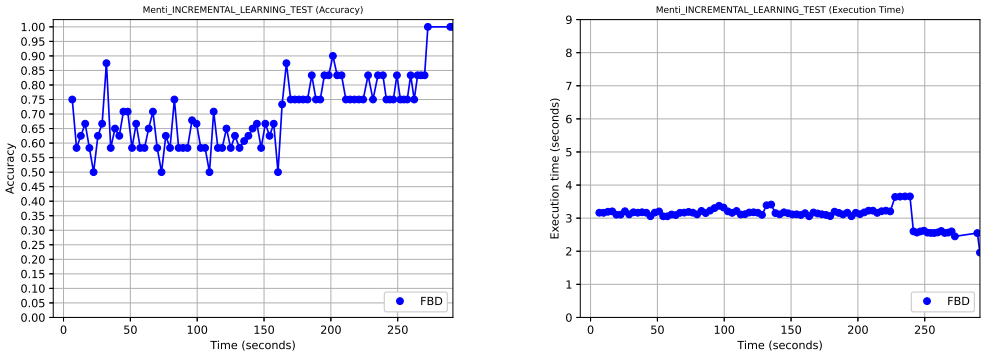


Figure 5.23. Menti, P2P Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.24. Menti Botnet, Incremental Learning Test.

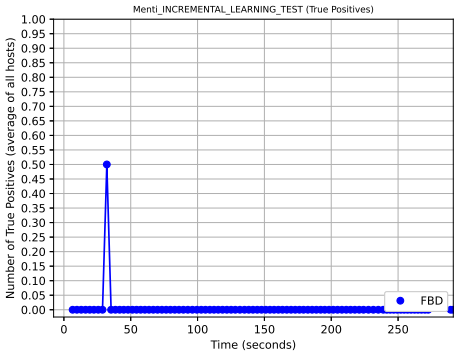
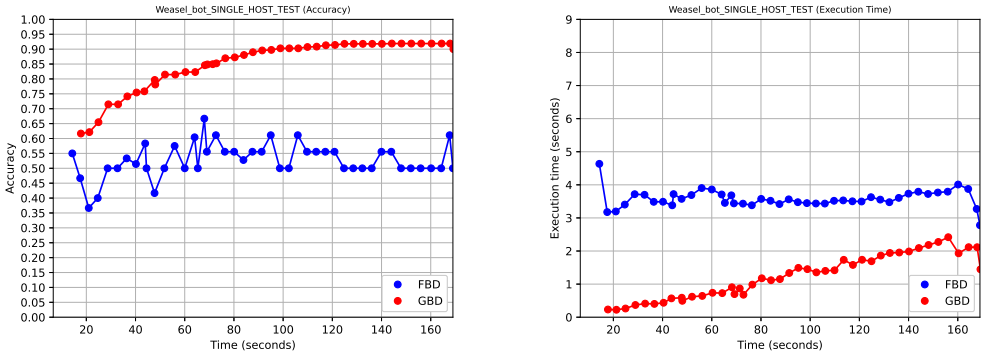


Figure 5.25. Menti, Incremental Learning Test, True Positives.

5.3.3 Results on unknown botnets

In the Menti botnet, we can see, as in the case of Neris, that the graph-based detection (gbd) accuracy of the *Single-host Test* (Figure 5.20a) grows over the time and, at 150 seconds, it settles at 0.95. Instead, in the case of flow-based detection (fbd), the accuracy varies from 0.33 to 1.0. The *P2P Test* was performed with only 2 peers communicating with the malicious host and its results are very similar to those of the *Single-host Test*. The accuracy of the Oracle is nearly 0.980 and the detection is performed in nearly 3.632 seconds; but despite a high accuracy, the True Positives value is 0.0. This is a clear indication that, if the number of peers composing the network is too small, the capability to build a meaningful graph to detect the botnet becomes a hard task, which is also an expected result from our proposal. Also, it could be an indication that additional features might be included in our proposed approach, to increment its detection capabilities. If we consider also the results of the *Incremental Learning Test* (Figure 5.24a and Figure 5.25), we can see that the Incremental Learning is not effective at improving the flow-based detection since the graph-based detection is not able to detect the botnet as seen previously.



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.26. Weasel Botnet, Single-host Test.

Considering the Weasel botnet, we can see that in the *Single-host Test*, graph-based detection (gbd) accuracy (Figure 5.26a) grows over the time and, at 120 seconds, it settles at 0.92. Instead, in the case of the flow-based detection (fbd), the accuracy varies from 0.36 to 0.67. The *P2P Test* was performed with 5 peers

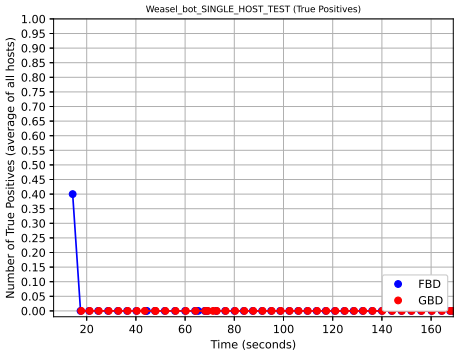
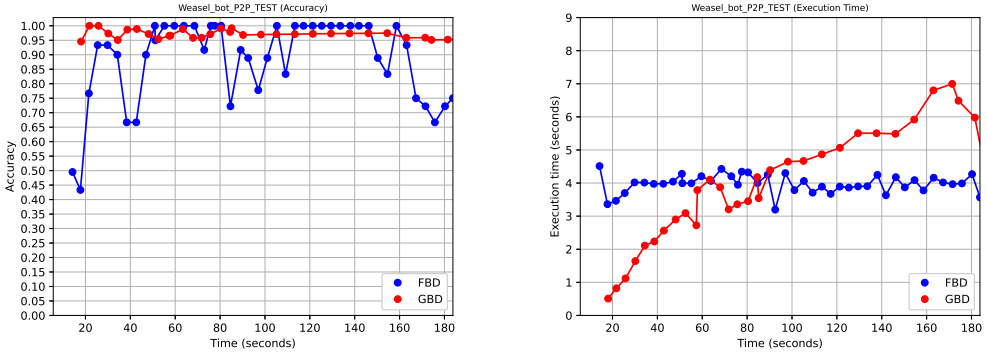


Figure 5.27. Weasel Botnet, Single-host Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.28. Weasel Botnet, P2P Test.

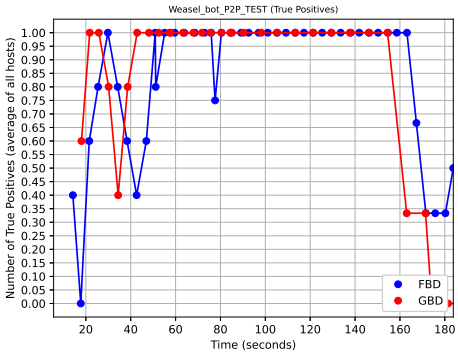
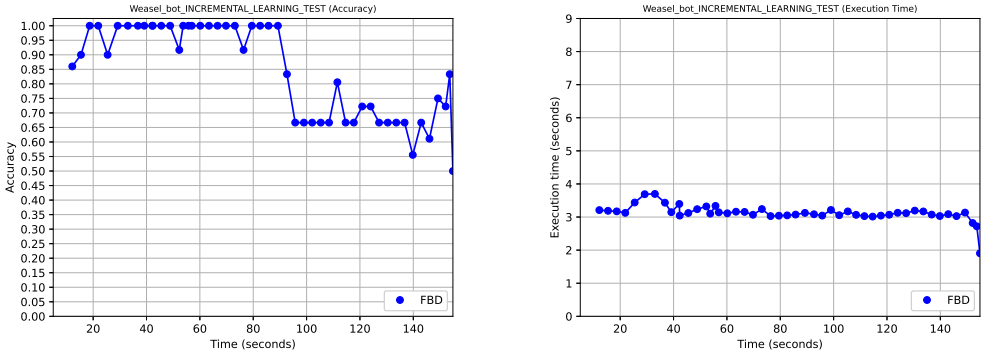


Figure 5.29. Weasel, P2P Test, True Positives

communicating with the malicious host. Comparing its results (Figure 5.28a) with those in *Single-host Test*, we can notice that the accuracy of the graph-based detection



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.30. Weasel Botnet, Incremental Learning Test.

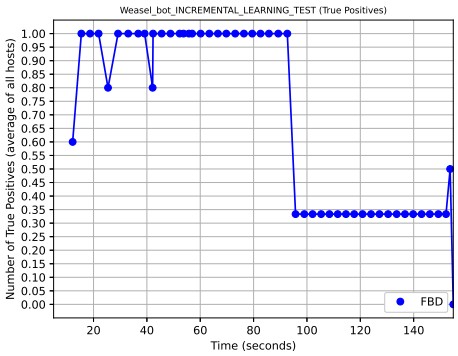
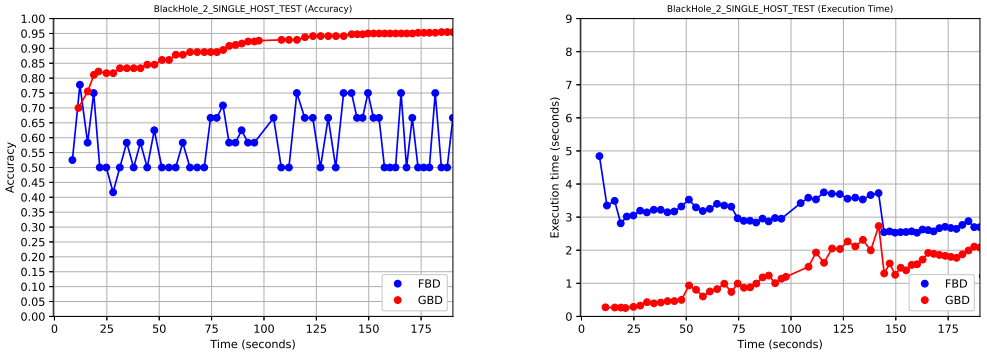


Figure 5.31. Weasel, Incremental Learning Test, True Positives.

is near 0.95 since $Time = 20s$ and, thanks to incremental learning, the accuracy of the flow-based detection is higher. This means, as already explained, that the host's knowledge of the network traffic exchanged by the all peers of the P2P Network communicating with the malicious host, as expected gives better results in detecting the malicious host. A peculiarity of the graph-based detection execution time in *P2P Test* (Figure 5.28b) is that it grows rapidly over time compared to other type of Botnet (for example, Neris). A further evidence of the incremental learning's effectiveness is shown comparing the Flow-based detection True Positives in the *Single-host Test* (Figure 5.27) with those in *Incremental Learning Test* (Figure 5.31): in fact, the incremental learning improves the accuracy of flow-based analysis on the samples of botnets already analyzed in the past by updating everytime the

flow-based training dataset. The only peculiarity is that the True Positives value in *Incremental Learning Test* is high until $Time \leq 80$. We can perform another evaluation of our method comparing the results of the *P2P Test* (Figure 5.28 and Figure 5.29) with the "Oracle" results. The accuracy of the Oracle is nearly 0.970, the True Positives has a value of 0.6 and the Oracle detection is performed in nearly 2.664 seconds. We can clearly state that the Oracle is more accurate and responsive than our method in "P2P mode" (i.e. when it performs the analysis communicating with others peers of the P2P Network) if we consider a botnet the host has never seen before. Instead, as shown in *Incremental Learning Test* results (Figure 5.30a and Figure 5.31), the analysis of samples of previously analyzed botnets causes an increase in the flow-based method accuracy that overcomes (until $Time \leq 80$), thanks to incremental learning, the graph-based accuracy of the Oracle. The execution time of the graph-based analyses in the *P2P test* (Figure 5.28b) is slightly greater than that of flow-based analyses in *Incremental Learning Test* (Figure 5.30b) making the general approach slightly less responsive also when analyzing samples of already analyzed botnets.



(a) Accuracy.

(b) Training Execution Time.

Figure 5.32. BlackHole Botnet, Single-host Test.

Considering the BlackHole botnet, we can see that in the *Single-host Test*, graph-based detection (gbd) accuracy (Figure 5.32a) behaves very similarly to the Neris Botnet growing over the time and, at 150 seconds, it settles at 0.95. Instead, in the case of the flow-based detection (fbd), the accuracy varies from 0.40 to 0.75.

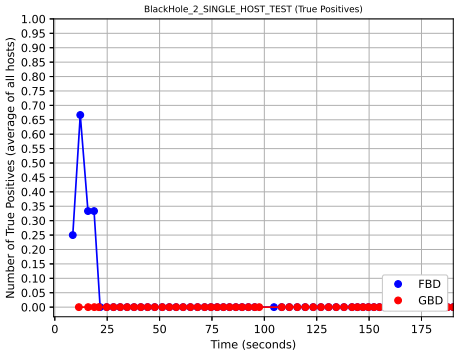
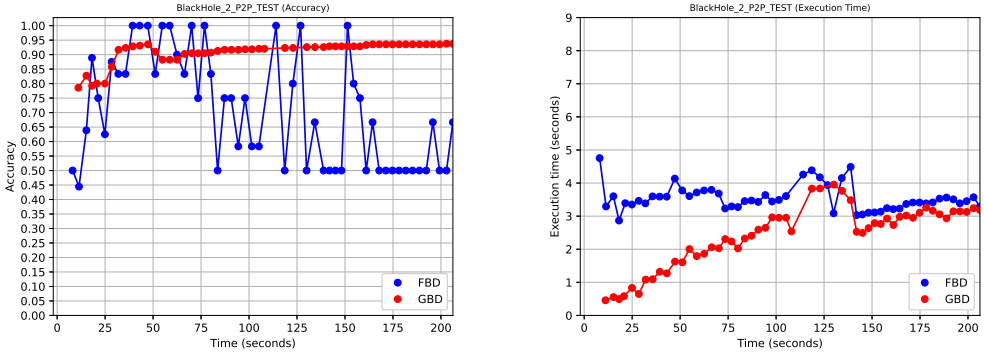


Figure 5.33. BlackHole Botnet, Single-host Test, True Positives



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.34. BlackHole Botnet, P2P Test.

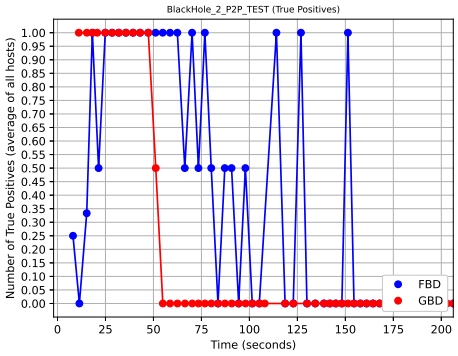
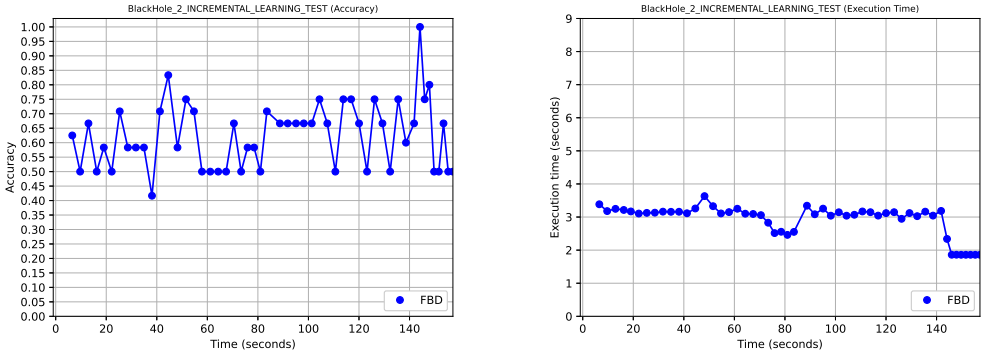


Figure 5.35. BlackHole, P2P Test, True Positives

The *P2P Test* was performed with 4 peers communicating with the malicious host and comparing its results (Figure 5.34a) with those in *Single-host Test*, we can



(a) Accuracy. **(b)** Training Execution Time.

Figure 5.36. BlackHole Botnet, Incremental Learning Test.

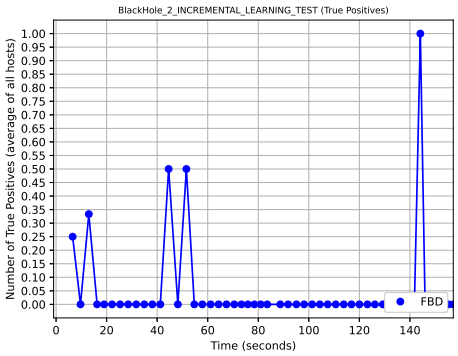


Figure 5.37. BlackHole, Incremental Learning Test, True Positives.

notice that the accuracy of the graph-based detection is near 0.90 since $Time = 26s$ and, thanks to incremental learning, the accuracy of the flow-based detection is higher at some time. A peculiarity is that the host's knowledge of the network traffic exchanged by the all peers of the P2P Network communicating with the malicious host, gives irregular results in detecting the malicious host (True Positives), as we can see in Figure 5.35. For this irregularity, Incremental Learning loses its effectiveness compared to the Flow-based detection True Positives in the *Single-host Test* (Figure 5.33) with those in *Incremental Learning Test* (Figure 5.37). The accuracy of the Oracle is nearly 0.843, the True Positives has a value of 0.5 and the Oracle detection is performed in nearly 1.058 seconds. We can notice that the accuracy of the Oracle is lower than that of other Botnets and also the performance

in detecting the malicious host is not high.

Chapter 6

Conclusions

In this work, we have shown an approach using flow-based and graph-based methods in a distributed system, which complies with our objectives: online detection and mitigation. As we have seen in chapter 5, the distributed approach gives good results with some types of Botnets which attack a sufficient number of peers in the P2P network, instead other types of Botnets which attack few peers, may be difficult to detect. Another good result is about Incremental Learning: it allows to detect properly some bots through the flow-based detection only if those bots have already been previously detected by the graph-based detection which has updated the flow-based training dataset. In future work, we can consider to improve the approach making it able to detect more botnets than one simultaneously by detecting the sub-graphs for each specific botnets from the main graph (built from the captured total P2P traffic) through Machine Learning techniques like clustering. Moreover, we can study more accurately the variation of the graph-based accuracy with respect to flow-based detection in a variable P2P Network (where the peers may become inactive over time). To this end, it is necessary to handle the ever increasing graph on which to perform the graph-based analysis in order to improve the performance of the approach.

Bibliography

[1] Abbas Abou Daya, Mohammad Salahuddin, Noura Limam, and R. Boutaba. Botchase: Graph-based bot detection using machine learning. *IEEE Transactions on Network and Service Management*, PP, 02 2020.

[2] Magnus Almgren and Wolfgang John. Tracking malicious hosts on a 10gbps backbone link. In Tuomas Aura, Kimmo Järvinen, and Kaisa Nyberg, editors, *Information Security Technology for Applications*, pages 104–120, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[3] S. Arshad, M. Abbaspour, M. Kharrazi, and H. Sanatkar. An anomaly-based botnet detection approach for identifying stealthy botnets. In *2011 IEEE International Conference on Computer Applications and Industrial Electronics (ICCAIE)*, pages 564–569, 2011.

[4] Simon Bernard, Laurent Heutte, and Sébastien Adam. A study of strength and correlation in random forests. In *International Conference on Intelligent Computing*, pages 186–191. Springer, 2010.

[5] E. Biglar Beigi, H. Hadian Jazi, N. Stakhanova, and A. A. Ghorbani. Towards effective feature selection in machine learning-based botnet detection approaches. In *2014 IEEE Conference on Communications and Network Security*, pages 247–255, 2014.

[6] James R Binkley and Suresh Singh. An algorithm for anomaly-based botnet detection. *SRUTI*, 6:7–7, 2006.

[7] Botnet dataset, 2014. `https://www.unb.ca/cic/datasets/botnet.html`, last accessed on 2021-01-13.

[8] Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 10 2001.

[9] D. Calavera and L. Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. O'Reilly Media, 2019.

[10] Ruidong Chen, Weina Niu, Xiaosong Zhang, Zhongliu Zhuo, and Fengmao Lv. An effective conversation-based botnet detection method. *Mathematical Problems in Engineering*, 2017:1–9, 04 2017.

[11] H. Choi, H. Lee, H. Lee, and H. Kim. Botnet detection by monitoring group activities in dns traffic. In *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, pages 715–720, 2007.

[12] Bpf and xdp reference guide, 2020. `https://github.com/cilium/cilium/blob/v1.9/Documentation/bpf.rst`, last accessed on 2021-01-13.

[13] Charles D Cranor, Emden Gansner, Balachander Krishnamurthy, and Oliver Spatscheck. Characterizing large dns traces using graphs. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 55–67, 2001.

[14] Lorrie Faith Cranor and Brian A LaMacchia. Spam! *Communications of the ACM*, 41(8):74–83, 1998.

[15] David Dagon. Botnet detection and response. In *OARC workshop*, volume 2005, 2005.

[16] J. Demarest. Taking down botnets, 07 2014. `https://www.fbi.gov/news/testimony/taking-down-botnets`, last accessed on 2021-01-13.

[17] Gameover zeus botnet disrupted, 06 2014. `https://www.fbi.gov/news/stories/gameover-zeus-botnet-disrupted`, last accessed on 2021-01-13.

[18] M. Feily, A. Shahrestani, and S. Ramadass. A survey of botnet and botnet detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273, 2009.

[19] Abe Fettig and Glyph Lefkowitz. *Twisted network programming essentials.* " O'Reilly Media, Inc.", 2005.

[20] Nsis.botnet, 12 2017. `https://www.fortiguard.com/encyclopedia/ips/45139/nsis-botnet`, last accessed on 2021-01-13.

[21] D. Garant and W. Lu. Mining botnet behaviors on the large-scale web application community. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 185–190, 2013.

[22] Sebastián García, Alejandro Zunino, and Marcelo Campo. Survey on network-based botnet detection methods. *Security and Communication Networks*, 7(5):878–903, 2014.

[23] J. Göbel and T. Holz. Rishi: Identify bot contaminated hosts by irc nickname evaluation. In *HotBots*, 2007.

[24] graph-tool documentation. `https://graph-tool.skewed.de/static/doc/centrality.html`, last accessed on 2021-01-13.

[25] B. Gregg. *BPF Performance Tools.* Addison-Wesley Professional Computing Series. Pearson Education, 2019.

[26] G. Gu, V. Yegneswaran, P. Porras, J. Stoll, and W. Lee. Active botnet probing to identify obscure command and control channels. In *2009 Annual Computer Security Applications Conference*, pages 241–253, 2009.

[27] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *Proceedings of the 17th Conference on Security Symposium*, page 139–154. USENIX Association, 2008.

[28] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[29] Yutaro Hayakawa. ebpf implementation for freebsd. In *BSDCan 2018.* The BSD Conference, 2018.

[30] K. J. Higgins. The world's biggest botnets, 11 2007. `https://www.darkreading.com/the-worlds-biggest-botnets-/d/d-id/1129117`, last accessed on 2021-01-13.

[31] Chien-Hau Hung and Hung-Min Sun. A botnet detection system based on machine-learning using flow-based features. In *Proceedings of the Twelfth International Conference on Emerging Security Information, Systems and Technologies, SECURWARE*, pages 122–127, 2018.

[32] Laheeb Mohammed Ibrahim and Karam H Thanon. Analysis and detection of the zeus botnet crimeware. *International Journal of Computer Science and Information Security*, 13(9):121, 2015.

[33] Muhammad Imam, Manjinder Paul Nir, and Ashraf Matrawy. A survey on botnet architectures, detection and defences. *International Journal of Network Security*, 17, 01 2014.

[34] Wolfgang John and Sven Tafvelin. Differences between in-and outbound internet backbone traffic. In *TERENA Networking Conference (TNC)*, 2007.

[35] Ahmad Karim, Rosli Bin Salleh, Muhammad Shiraz, Syed Adeel Ali Shah, Irfan Awan, and Nor Badrul Anuar. Botnet detection techniques: review, future trends, and issues. *Journal of Zhejiang University SCIENCE C*, 15(11):943–983, Nov 2014.

[36] Kaspersky. What is a botnet? `https://www.kaspersky.com/resource-center/threats/botnet-attacks`, last accessed on 2021-01-13.

[37] Zeus virus. `https://www.kaspersky.com/resource-center/threats/zeus-virus`, last accessed on 2021-01-13.

[38] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *Forest*, 23, 11 2001.

[39] Lei Liu, Songqing Chen, Guanhua Yan, and Zhao Zhang. Bottracer: Execution-based bot-like malware detection. In *International Conference on Information Security*, pages 97–113. Springer, 2008.

[40] Mohammad M Masud, Tahseen Al-Khateeb, Latifur Khan, Bhavani Thuraisingham, and Kevin W Hamlen. Flow-based identification of botnet traffic by mining multiple log files. In *2008 first international conference on distributed framework and applications*, pages 200–206. IEEE, 2008.

[41] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

[42] Ctu-malware-capture-botnet-42, 2016. `https://mcfp.weebly.com/ctu-malware-capture-botnet-42.html`, last accessed on 2021-01-13.

[43] I. C Mogotsi. Christopher d. manning, prabhakar raghavan, and hinrich schütze: Introduction to information retrieval. *Information Retrieval*, 13:192–195, 04 2010.

[44] Christian FA Negre, Uriel N Morzan, Heidi P Hendrickson, Rhitankar Pal, George P Lisi, J Patrick Loria, Ivan Rivalta, Junming Ho, and Victor S Batista. Eigenvector centrality for characterization of protein allosteric pathways. *Proceedings of the National Academy of Sciences*, 115(52):E12201–E12208, 2018.

[45] Virut botnet, 12 2018. `https://digital.nhs.uk/cyber-alerts/2018/cc-2829`, last accessed on 2021-01-13.

[46] White Ops. 9 of history's notable botnet attacks, 05 2018. `https://www.whiteops.com/blog/9-of-the-most-notable-botnets`, last accessed on 2021-01-13.

[47] Abdurrahman Pektaş and Tankut Acarman. Botnet detection based on network flow summary and deep learning. *International Journal of Network Management*, 28(6):e2039, 2018. e2039 nem.2039.

[48] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Proceedings of LISA '99: 13th Systems Administration Conference*, pages 229–238, 1999.

[49] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 01, pages 209–217, 2018.

[50] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux socket filtering aka berkeley packet filter (bpf). `https://www.kernel.org/doc/Documentation/networking/filter.txt`, last accessed on 2021-01-13.

[51] Kapil Sinha, Arun Viswanathan, and Julian Bunn. Tracking temporal evolution of network activity for botnet detection. *arXiv preprint arXiv:1908.03443*, 2019.

[52] M. Stevanovic and J. M. Pedersen. An efficient flow-based botnet detection using supervised machine learning. In *2014 International Conference on Computing, Networking and Communications (ICNC)*, pages 797–801, 2014.

[53] Elizabeth Stinson and John C Mitchell. Characterizing bots' remote control behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 89–108. Springer, 2007.

[54] Ihsan Ullah, Naveed Khan, and Hatim A Aboalsamh. Survey on botnet: Its architecture, detection, prevention and mitigation. In *2013 10th IEEE International Conference on Networking, Sensing and Control (ICNSC)*, pages 660–665. IEEE, 2013.

[55] Jignesh Vania, Arvind Meniya, and HB Jethva. A review on botnet and detection technique. *International Journal of Computer Trends and Technology*, 4(1):23–29, 2013.

[56] Wei Wang, Yaoyao Shang, Yongzhong He, Yidong Li, and Jiqiang Liu. Botmark: Automated botnet detection with hybrid analysis of flow-based and graph-based traffic behaviors. *Information Sciences*, 511:284 – 296, 2020.

[57] Kui Xu, Danfeng Yao, Qiang Ma, and Alexander Crowell. Detecting infection onset with behavior-based policies. In *2011 5th International Conference on Network and System Security*, pages 57–64. IEEE, 2011.

[58] H. R. Zeidanloo, Mohammad Jorjor Zadeh Shooshtari, Payam Vahdani Amoli, M. Safari, and M. Zamani. A taxonomy of botnet detection techniques. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 2, pages 158–162, 2010.

[59] Justin Zhan, Sweta Gurung, and Sai Phani Krishna Parsa. Identification of top-k nodes in large networks using katz centrality. *Journal of Big Data*, 4(16), 05 2017.

[60] David Zhao, Issa Traore, Bassam Sayed, Wei Lu, Sherif Saad, Ali Ghorbani, and Dan Garant. Botnet detection based on traffic behavior analysis and flow intervals. *Computers & Security*, 39:2–16, 11 2013.