FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Detection of IoT Malware in Computer Networks
Student:	Bc. Daniel Uhříček
Supervisor:	Ing. Karel Hynek
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2021/2022

Instructions

Get acquainted with network monitoring methods based on deep packet inspection and (extended) IP flows. Analyze the area of IoT malware and its behavior on the computer network, focusing on detection possibilities. Explore available IoT datasets; focus on significant characteristics of malware communication. Design an algorithm for automatic detection of IoT malware presence in the network based on observed network traffic. Develop a software prototype capable of processing real network traffic in the NEMEA system. Test and evaluate the prototype with the created dataset and data provided by the supervisor of this thesis.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 6 January 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Detection of IoT Malware in Computer Networks

Bc. Daniel Uhříček

Department of Information Security
Supervisor: Ing. Karel Hynek

May 6, 2021

Acknowledgements

I would like to express my sincere gratitude to Ing. Karel Hynek for his guidance, advice, feedback, and all the devoted hours. I thank Dr. Armin Wasicek, who brought me to the research part of network security and supported the smooth ongoing of my work. I would also like to thank my girlfriend for her immense patience, especially during the last weeks of writing this thesis; as well as my parents, who provided a lot of support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 6, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Uhříček, Daniel. *Detection of IoT Malware in Computer Networks.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato diplomová práce se zabývá problematikou IoT malwaru a možnostmi jeho detekce v počítačových sítích na úrovni monitoringu síťových toků. V práci identifikujeme klíčové aspekty chování IoT malwaru a odděleně prezentujeme možnosti jejich řešení. Práce navrhuje nový přístup pro detekce nakažených zařízení za použití kombinace síťových indikátorů. Navrhovaná metoda byla implementovaná ve formě softwarového prototypu, schopného zpracovávat reálný síťový provoz v NEMEA systému. Finální řešení bylo vyhodnoceno na anonymizovaných záchytech a aktuálních vzorcích malwaru.

Klíčová slova IoT malware, botnet, monitorování síťových toků, C&C komunikace, detekce anomálií

Abstract

This master thesis deals with the problematics of IoT malware and the possibilities of its detection in computer networks using flow-based monitoring concepts. We exhibit solutions for each of the identified critical aspects of IoT malware network behavior separately. Furthermore, we propose a novel method to discover infected devices using a combination of network indicators. The proposed detection method was implemented in the form of a software prototype capable of processing real network traffic as part of the NEMEA system. The final solution was evaluated both on anonymized captures and up-to-date malware samples.

Keywords IoT malware, botnet, flow monitoring, C&C communication, anomaly detection

Contents

List of Figures

List of Tables

Introduction

Internet of Things (IoT) is becoming incredibly popular even in the consumer sector. Data [1] collected from 16 million user-initiated network scans show that 71 % of North American households and 57 % of Western European households have an IoT device. The majority of recognized devices implement a service on top of TCP or UDP transport protocols, leading with UPnP, HTTP, and mDNS. Additionally, 7.1 % of the devices allow Telnet communication. Sadly, widespread brands of consumer-end IoT devices often comprise significant security vulnerabilities.

Low-security standards, weak default credentials, and unpatched remote code execution vulnerabilities lead to the rapid spread of IoT malware. Since many IoT devices build their firmware on top of embedded Linux, we recognize IoT malware as malicious Linux software targeting particular IoT devices. Compared to other platforms, IoT malware analysis is still underestimated. Historically, security companies were focused on other operating systems such as Windows or Android, having already established anti-malware solutions. [2].

The primary goal of this thesis is to research the network behavior of prevalent IoT malware families and apply the acquired knowledge to design a mechanism automatically detecting IoT malware presence in computer networks. We aim to raise the situational awareness of network monitoring operators, identifying infected hosts, command-and-control (C&C) servers, or ongoing attacks. The mechanism should be transformed into a software prototype capable of processing real network traffic in NEMEA – modular, flow-based network detection system maintained by CESNET.

The thesis is structured into six chapters. Chapter 1 discusses network security monitoring concepts. It reviews methods of deep packet inspection, pattern matching, application-layer monitoring, and IP flows. Chapter 2 introduces prominent IoT malware families. Furthermore, it explains the automated IoT malware analysis procedures, which are later used to identify critical points of malicious network behavior. Chapter 3 examines publicly available IoT datasets and describes the process of custom dataset prepa-

ration. Chapter 4 proposes a method to detect infected IoT devices in computer networks, targeting different aspects of IoT malware behavior separately. Chapter 5 outlines the implementation and deployment of a proof-of-concept detection system. Finally, Chapter 6 sums up the evaluation results on the gathered IoT datasets, long benign captures provided by CESNET, and up-to-date malware traffic supplied by Avast Software.

Chapter 1

Network Security Monitoring

This chapter discusses the current principles of network security monitoring and its applications in network intrusion detection systems. We review traditional deep packet inspection (DPI) methods executed at the packet level, introducing the concepts of pattern matching and signatures. After stating the particular drawbacks of the DPI approach, we examine the monitoring on the higher-level view of abstraction – IP flows. We describe flow-based monitoring components, protocols, and recent expansions.

1.1 Deep Packet Inspection

DPI has many uses both in network monitoring and network security. Internet Service Providers (ISPs) and network administrators are concerned about their network's efficiency and must implement bandwidth management to limit certain types of internet traffic. The government may have requirements to see into traffic for law enforcement and surveillance purposes. Companies must have a way to enforce internal policies and monitor security incidents [3]. Considering security, DPI is a vital part of many network intrusion detection systems (NIDS) that deal with application layer information. Utilizing DPI, we analyze the packet header information and the payload to recognize any desired patterns. We will further consider only open-source DPI NIDS – Snort[1], Suricata[2] and Zeek[3] – so we can assure ourselves of the applied principles and algorithms.

1.1.1 Pattern Matching

Stages of a generic DPI architecture are shown in Figure 1.1. The second to last step, detection or recognition of the packet payload, is distinctive to

[1] https://www.snort.org
[2] https://suricata-ids.org
[3] https://zeek.org/

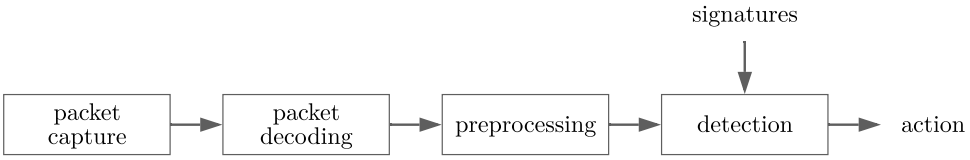

Figure 1.1: A generic architecture of signature-based DPI system as described [5]. Packets are captured, preprocessed, and matched against loaded patterns. Everything results in the final action specified by the signature – such as throwing an alert or dropping subsequent packets.

DPI. The basic approach to payload recognition is through sequence matching, where popular matching algorithms such Aho-Corasick are used [3, 4]. Where string detections are not sufficient, regular expressions are typically used. They are accepted with finite state automata, either non-deterministic finite automata or deterministic finite automata. However, processing each packet with regular expressions would significantly limit the system's throughput [3].

To speed up the evaluation, DPI engines do not match regular expressions on all packets. Instead, they adopt a two-stage matching algorithm called prefilter-based matching (see Figure 1.2. However, as noted by Wang et al. [4], two major weak points in this approach exist:

- The prefilter-based matching is prone to wrong manual choice prefilter multistrings. Improperly chosen prefilter string, a string that could be immensely common in flowing network traffic, will cause that in the end, most packets will also be evaluated with a subsequent regular expression.
- Two separate tasks – multistring matching and regular expression matching, are redundant to some degree. When a regular expression is evaluated, duplicate matching occurs for the corresponding string keyword.

To address these issues, Wang et al. [4] define a new matching engine called Hyperscan. Hyperscan removes redundant operations by integrating string matching as part of regular expression matching. The base technique of this integration – decomposition-based matching – decomposes patterns into a string and regular expressions that have to be matched in order. For the automatic decomposition of regular expressions, the authors used graph-based decomposition techniques. Moreover, all their algorithms are implemented to leverage SIMD (Single Instructions Multiple Data) instructions of modern CPUs. Hyperscan was also open-sourced[4] and made available in Suricata and Snort as an alternative pattern matching engine.

[4] `https://github.com/intel/hyperscan`

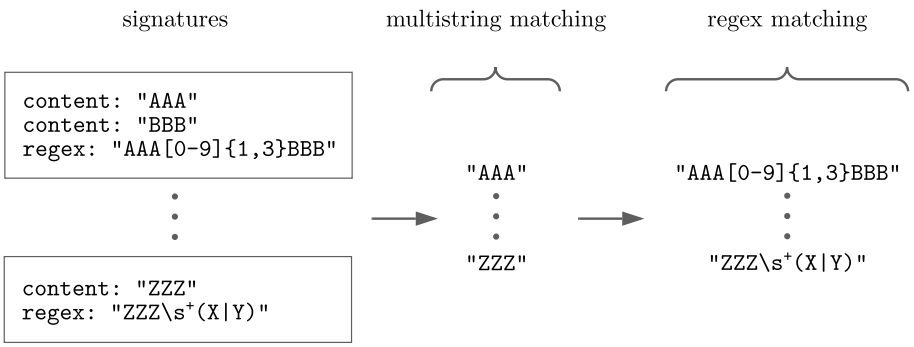

Figure 1.2: Prefilter-based pattern matching [4] – one of available implementations in Snort and Suricata. In the first stage, engine runs fast multistring matching (using e.g. Aho-Corasick algorithm). In the second stage, single regular expression patterns are evaluated.

1.1.2 Signatures

Patterns or signatures describe specific characteristics of network communication [3, 6]. They can express typical network patterns of attack, malware C&C channel, or peer-to-peer communication. Signatures can be created either manually by domain experts or by using some rule-generating algorithms. Nevertheless, to produce a new signature, the attack must be first recognized, and there is a delay between spotting a specific malicious communication the first time and creating its signature.

Languages for writing signatures tend to be system-specific. Nonetheless, the format of Snort's and Suricata's signatures is similar, and in many cases, the signatures can be converted from one system to another. Additionally, their main principles and system architecture are the same. They differ in lower-level implementations (such as Suricata leveraging multi-threading) [7]. Zeek also accepts its signatures (see Figure 1.3). On the contrary, it is comparably less dependent on the traditional ways of writing signatures. Its main purpose lies in protocol recognition and custom domain-specific scripting language [5]. Out-of-the-box, it generates more than 60 logs [8]. Apart from protocol-related Zeek logs (`dns.log`, `dhcp.log`, `ftp.log`, `http.log`, etc.), the most important logs are `conn.log` (summarizing TCP, UDP, and ICMP connections), `files.log` (extracted downloaded files), and `signatures.log` (logging matched signatures). The list of logs can be further broadened using the mentioned Zeek scripting language.

Snort's and Suricata's signatures are both separated into a header and a body. The header contains information about the decided action (raising an alert or dropping consequent packets), specification of a protocol, IP addresses, ports, and direction. Suricata, compared with Snort, supports the recognition of many application-level protocols (that are written in the header

```
signature malware-signature {
    ip-proto == tcp
    dst-port == 80
    payload /.*malware/
    event "Found malware string in payload!"
}
```

Figure 1.3: Zeek signature format allows defining regular expression to search for particular content sequences.

```
alert tcp [!10.0.10.0/24, !10.0.20.0/24] any -> any 42 \
(content:"malware"; msg:"Found malware string in payload!")
```

Figure 1.4: Example signature compatible with Snort and Suricata. The signature starts with an action, followed by a protocol, and two combinations of IP addresses and ports. We see here that source IP addresses can also be specified for example as list of negated IP ranges or with a special keyword `any`.

part). The body is composed of several keywords and values. Suricata documentation [7] specifies separate keywords for each targeted protocol (such as IP keywords, HTTP keywords, or DNS keywords) and meta-information keywords (such as an alert message or signature id). Figure 1.4 shows an example of a simple rule.

1.1.3 Application Layer Monitoring

DPI systems, in theory, are not limited either by the amount or the diversity of application protocols to support. They are often extended with new protocol parsers if the corresponding protocol poses valuable information. Popular baseline choices for many tasks (including malware detection) are DNS, HTTP, and TLS.

DNS

Domain Name System (DNS) is a mechanism providing names for different entities (e.g., hosts or services) on the network and their translation [9]. Besides header information, we can dig into four sections of DNS packets: question section, answer section, authority section, and additional section [10]. The question section, as the name suggests, describes the question to the DNS server. Answer section, authority section, and additional section are all lists of resource records. Each resource record consists of name, type, class, TTL, length, and data. Eventually, we can filter out the resource record types we

are interested in, such as IPv4 and IPv6 addresses (`A` and `AAAA` types), canonical names (`CNAME` type), authoritative name servers (`NS` type), or additional text information (`TXT` type).

DNS information can be used for domain blacklisting, although more involved methods of DNS security were described. For example, in [11], authors describe methodologies to detect botnets' C&C using DNS query activities. Their framework (Botnet Group Activity Detector) extracts both query features and answer features from respective resource records, applies lexicographical methods to distinguish text patterns, and finally compares the domains' similarities.

HTTP

Hypertext Transfer Protocol (HTTP) is a request-response type of protocol that manipulates web resources [12]. Resources are addressed by their URI (Uniform Resource Identifier), which are present in HTTP requests. Apart from URI, HTTP requests contain request method, HTTP version, headers, and encoded data [12]. In HTTP responses, we may be interested in the status code, headers, and encoded data. HTTP monitoring poses benefits of detecting anomalous User-Agents, malicious requests to vulnerable APIs, or generic malware data present in HTTP payloads. All of this is usually addressed by pattern matching mechanisms described earlier in this chapter.

TLS

Transport Layer Security (TLS) allows two endpoints to establish and use a secure communication channel. This channel should provide authentication, confidentiality, and integrity. Suppose we neglect the case where the monitoring device intercepts TLS communication and acts as a man-in-the-middle. In that case, the only part of TLS valid for a DPI system is the handshake protocol. TLS handshake starts right after TCP three-way handshake, initiated by the client with Client Hello message. The server responds with Server Hello message if it can accept the client's set of attributes. After Client Hello and Server Hello messages, four attributes are agreed upon (protocol version, session ID, cipher suite, and compression method). Then, the server sends its certificate. Information up to this point is expected to be examined in a DPI system.

TLS certificates may be collected and blacklisted if they are linked to malicious activity (similarly to blacklisting IP addresses or domains). Another method developed by Salesforce is JA3 and JA3S TLS fingerprinting [13], which they open-sourced in 2017. It builds on the fact that different TLS applications (clients or servers) may support a different set of attributes for establishing a TLS connection. These attributes are extracted from Client Hello and Server Hello messages, serialized, and hashed into JA3 and JA3S

hashes correspondingly. Values extracted from Client Hello messages are version, cipher suite, extension, elliptic curves, and elliptic curve point formats. Values are concatenated into a column-separated string and hashed applying the MD5 algorithm. The same goes for JA3S, but the values extracted from Server Hello messages are only: version, cipher suite, and extensions.

1.1.4 Open-source Rulesets

Individual signatures are grouped into bigger rulesets. Rulesets must be frequently updated to address new network security threats. Writing signatures is a time-consuming task that also requires dedicated personnel to handle false positives. Hence, it is commonly outsourced, and signatures are bought like any other security product. We examined open-source Suricata rulesets, assuming that the results would be similar for competing alternatives in Snort.

The biggest available ruleset is Emerging Threats[5]. Emerging Threats ruleset is released by Proofpoint, embodying two different components – ET PRO (paid feed) and ET OPEN (open-source). The latter is licensed under the BSD license (most of the rules) and the GPLv2 license. It contains rules for various categories such as malware, phishing, exploits, or malicious User-Agents. ET OPEN contains in total more than 20 000 rules. The most frequently used attributes in this ruleset can be seen in Table 1.1. Another interesting ruleset, SSLBL[6], is provided by abuse.ch. They gather information about malicious TLS connections and produce blacklists for both the certificate fingerprints and JA3 fingerprints. Their list of malicious certificates is regenerated every five minutes, updated with the newly acquired intelligence. Their blacklist can be downloaded in the form of Suricata rules.

1.1.5 Disadvantages of DPI Approach

There exist disadvantages of using per-packet DPI methods in high-speed computer networks. Firstly, DPI methods become problematic when the traffic is encrypted. We mentioned the use case of monitoring TLS by extracting information about certificates. Equally, it could be applied to get at least some insights on the HTTPS traffic. Another approach would be intercepting HTTPS traffic and effectively conducting a man-in-the-middle attack, primarily feasible only in corporate networks where the company can enforce the installation of certificates issued by their certificate authorities. We must also take into account the continuous traffic encryption trend with protocols such as DNS over HTTPS (DoH) [14] or QUIC [15]. Secondly, on the ISP backbone with 100 Gbps bandwidth, packet processing software solutions are not viable, and the processing is often accelerated in hardware using specialized network interface cards.

[5] https://rules.emergingthreats.net/
[6] https://sslbl.abuse.ch/

rules	buffer
22 100	`http.uri`
8867	`pkt_data`
4963	`file.data`
3968	`http.method`
3418	`http.request_body`
2605	`http.header_names`
2353	`dns.query`
1789	`http.user_agent`
1234	`http.header`
859	`http.host`
499	`tls.cert_subject`
435	`tls.sni`

Table 1.1: Most frequent attributes used in ET OPEN Suricata rules. Note that rules may use combination of attributes.

1.2 IP Flows

The concept of IP flows grants us higher-level views on network traffic, focusing mainly on header information and meta-information. In comparison with the DPI approach, which works on a packet level, in flow monitoring, we are granted an aggregated abstraction called *flow*. This approach to network security monitoring leads to a significant reduction of analyzed data [16].

An IP flow is defined as *"a set of IP packets passing an observation point in the network during a certain time interval, such that all packets belonging to a particular flow have a set of common properties"* [16, 17]. Such a set of properties are commonly denoted as *flow key*. Typical values of a flow key are source IP address, destination IP address, source port, destination port, and IP protocol. This definition might also resemble the description of a TCP connection, but note that the concept of IP flows was originally meant as unidirectional – for each direction of traffic "flowing" (from source to destination and vice versa), there are generated two individual flows [17]. Later, bidirectional flow records were defined for use cases where tracking directions individually was not feasible. In such a case, modification of the flow record cache must be made to identify and distinguish forward and backward directions.

IP flow monitoring is widespread in high-speed computer networks for its scalable architecture. Two of its necessary components are the flow exporter and the flow collector. Those two have one-way communication over a supported flow export protocol. A generic architecture of the flow monitoring process is described in Figure 1.5.

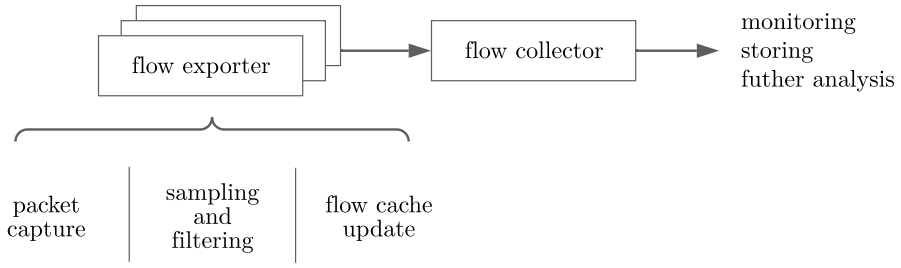

Figure 1.5: Generic flow monitoring architecture as presented in [16, 17]. Flow exporter (1) captures and observes packets; (2) samples and filters packets if necessary; (3) updates metered values or creates a new record in the flow cache; (4) exports flow records over an export protocol. Flow collector processes exported flow records coming from a scalable number of flow exporters. Collected data is further stored and analyzed.

1.2.1 Export Protocols

In 1996, Cisco patented their flow export technology called NetFlow. Despite now being used heavily for IP network monitoring, it was initially invented for flow-based switching. Storing flow information in a so-called flow cache allowed to decrease the time of forwarding decisions as they were only made for the first packet of each flow. They later found out that the flow cache also holds valuable information, and they designed NetFlow to export the data stored inside the flow cache [16].

The two widely adopted NetFlow versions are NetFlow v5 and NetFlow v9, the latter being a significant extension supporting IPv6, MPLS, VLANs, and templates. Templates became a way to extend existing data formats and define custom fields to export. The support of templates can also be found in the second, more recent, flow export protocol – IPFIX. IPFIX was first specified in 2008, taking the basis from the NetFlow v9, supporting many new features such as a definition of structured data [16].

1.2.2 Exporter

Prior to flow exporter processing, there is sometimes considered a separated packet observation phase, which puts timestamps on individual packets and eventually does some sampling and filtering. Furthermore, we will assume that the packet observation is part of an exporter.

In a flow exporter (also known as flow probe), incoming packets are aggregated into flows, and in the form of flow records, their information is stored in a flow cache. Values of flow records are from a set of elements – IPFIX calls these elements Information Elements [16]. Traditionally Information Elements came from the header fields of network and transport layers. Nowadays, In-

formation Elements are not limited to any ISO/OSI layer (which means flow security monitoring can also be application-aware similarly to the DPI approach as it will be further discussed in Section 1.2.4). We already mentioned that some of the Information Elements are chosen as flow key values. During packet processing and their aggregation into flows, the hash of the flow key is calculated and looked up in the flow cache. Afterward, either a new record in a flow cache is created, or the flow record's values and counters are updated.

Flow record is exported in the following circumstances [17, 16]:

- Active timeout – The flow is still active, but it becomes longer than the threshold for the maximum time duration of a flow. The flow record is exported, and all its counters are reset within the flow cache.
- Inactive timeout – No packets belonging to this specific flow were captured for a period longer than the inactive (idle) threshold.
- End of the TCP connection – In the case of TCP, the flow record is exported whenever we see FIN or RST flags, as those signal the termination of the TCP connection.
- Flow cache memory collision – Flow cache has clearly a limited size, and recourses may get exhausted. Two different flows might have hash flow key collisions, and we must either export one of the flow records or solve it dynamically.

In [16], authors summarized both open-source and commercial flow exporters. Classified them based on supported protocols, their options, flow cache properties, and application-level awareness. Commercial solutions tend to aim at high-speed computer networks. Typical network vendors such as Cisco or Juniper have their commercial solutions. It is also worth mentioning Flowmon with their high-performance application-aware probes built on top of COMBO cards with FPGA boards to support 100 Gbit/s networks. In the category of open-source exporters, we will be most interested in ipfixprobe [18], bidirectional flow exporter, which is part of CESNET NEMEA system. However, in Section 1.2.4 we will also look at other exporters and their extensions that have been used in state-of-art flow analysis publications.

1.2.3 Collector

Flow collector receives exported flow records in NetFlow or IPFIX format. It preprocesses the incoming flow data and stores them or sends them for further analysis [16]. CESNET ipfixcol2 [19] is an implementation of a collector used in the NEMEA system. It can process both NetFlow and IPFIX and is highly extensible by plugins. As its preprocessing, it can anonymize IP addresses in flow records. It allows four storage formats: (1) FDS (Flow Data Storage) file

format for long-term storage, (2) JSON format, (3) Infstore (nfdump compatible) format, and (4) UniRec. The last one is used throughout all plugins in the NEMEA framework.

1.2.4 Extended IP Flows

We mentioned that NetFlow v9 and IPFIX both support templates. Thanks to this template architecture, we can extend the basic functionality and information contained within the exported flow records. RFC 5102 [20] describes many properties of the IPFIX information model. Apart from apparent IP and transport header fields, timestamps, and exported configuration; RFC categorizes:

- Derived packet properties – Information Elements derived from individual packets. Namely, it is IP payload length (packet length starting after the IP header), next hop IP addresses, BGP information, and MPLS information.
- Min-max flow properties – Results of minimum or maximum aggregate functions computed over all packet properties – minimum and maximum packet lengths, minimum and maximum TTL, etc.
- Per-flow counters – Integer values classified either as running counters or delta counters. Delta counters are later being zeroed out after each exporting process. Examples of counters are the total number of all packets, delta counter of all packets, the total number of SYN packets, the total number of PSH packets, etc.

It may also be desirable to extend flow monitoring to the application level. Indeed, making the monitoring aware of selected application protocols grants us additional valuable visibility and allows us to use known DPI techniques in the aggregated flow-level view. However, extensions of IP flows are not unified, and several open-source flow monitoring projects exist that extend IP flows in their own way.

Cisco Joy [21] aims to help with data analysis in networking research and security monitoring. It extracts data features and represents the features in JSON format. It incorporates sequences of packet lengths and arrival times, estimation of a byte probability distribution, and entropies. For application-level protocols, it analyzes DNS (domain names, addresses, TTLs), TLS (cipher suites, server certificate strings), and HTTP (header information plus the first eight bytes of HTTP payload).

Already mentioned CESNET's implementation of a bidirectional flow exporter, ipfixprobe [18], extends IP flows through a plugin system. Each plugin inherently also carries the basic fields of source and destination information (MAC addresses, IP addresses, ports), timestamps of the first and the last

packet, and, for both directions, number of packets, number of bytes, and TCP protocol flags. Analogously to Cisco Joy, it can export sequential information about the first n packets. Its `PSTATS` plugin exports a sequence of packet sizes, a sequence of timestamps, a sequence of packet directions, and a sequence of TCP flags. Similar to Cisco Joy's estimation of the distribution of packet sizes, ipfixprobe can export histograms (using `PHISTS` plugin) of packet sizes and packet inter-arrival times. From application-level protocols, it supports DNS, TLS, HTTP, NTP, SIP, SMTP, SSDP. Moreover, it is being actively developed (in the time period of writing this thesis, at least three new plugins were released).

Another representant of bidirectional flow exporters is YAF [22] (Yet Another Flowmeter). Apart from some shared features with the previously described exporters, YAF can calculate the entropy of the first n bytes of the payload, recognize and label some specific applications. Besides, the application recognition can be extended through regular expressions.

1.2.5 Flow-based Statistical Features

As mentioned for Cisco Joy and ipfixprobe, IP flows can carry generic values representing analytical information such as histograms of bytes or any time-related information. From this information, we can extract features independent of our ability to understand the application layer, gaining insights by eventually applying statistical or machine learning methods. Therefore, it is also applicable to the analysis of encrypted traffic where traditional DPI methods fail.

Authors of [23, 24] proposed a flow-based classification method for the analysis of encrypted VPN and Tor traffic. In both cases, they were using only time-based features. Their features are generated from bidirectional flows, where the direction is determined by the direction of the first incoming packet. They compute mean, minimum, maximum, and standard deviation for packet inter-arrival times (separately for arrival times of the forward traveling packets, backward traveling packets, and all packets neglecting the direction). They monitored active and idle states of the flow and computed their ratios. Finally, they computed some fixed-time statistics such as the number of packets per second and the number of bytes per second. Their tool for flow-based feature extraction, CICFlowMeter, is publicly available on GitHub[7]. In [25], authors were training deep autoencoders for botnet anomaly detection based on similar features. Furthermore, they added features calculated from packet sizes and calculated correlation coefficients to address traffic periodicity. Similar features were also used by authors of [26] used this approach to monitor C&C traffic, showing that time-based features can distinguish C&C flows from other background flows.

[7] `https://github.com/ahlashkari/CICFlowMeter`

Chapter 2

IoT Malware

Malware (malicious software) is software generally causing harm, developed for financial profit, cyber espionage, disruption, or educational purposes [10]. We can distinguish malware based on its behavior into categories such as viruses, worms, or backdoors. In the world of IoT malware, the dominating type of malware is a botnet [2]. Botnets intend to create a large network of infected devices (called bots or zombies), which are then controlled by a so-called command-and-control (C&C) server [27]. Attackers manage C&C servers, allowing their operators to send commands for individual actions. Botnet implementation differs on the level of available commands, their functionality, and the type of communication infrastructure. The two most common purposes of IoT botnets are distributed denial-of-service (DDoS) attacks and cryptocurrency mining. The range of desired behavior can be often extended since it is practical for attackers to implement botnet commands to download and execute any other payload. Infrastructure-wise, botnets can use existing communication channels as a C&C (such as IRC); or implement their own communication protocol – either client-server or peer-to-peer. This chapter focuses on the history, development, and current state of IoT malware, its analysis, and critical aspects of its network-related behavior.

2.1 IoT Botnet Families

The first large-scale comprehensive study of IoT malware was carried out by researchers from Eurocom and Cisco [2] in 2018. The study researched more than 10 000 unique malware samples using static and dynamic analysis. It showed that although IoT malware is not as complex as Windows malware, it is slowly adopting more advanced techniques. The samples are commonly packed with UPX or its modifications. Moreover, reverse engineers must deal with multiple non-x86 processor architectures (such as MIPS or ARM). Later study [28] from the year 2020 presented research on IoT malware families through their code similarity. The authors researched more than 93 000 sam-

ples, trying to reconstruct malware families' evolution and mutual relationships. All samples were submitted to VirusTotal and labeled with AVClass (a labeling tool to determine the most likely family name based on antivirus detections). Their data shows that the three most dominating families are Tsunami, Gafgyt, and Mirai. Further, we shortly introduce them together with other important IoT malware families that have been emerging throughout the years.

2.1.1 Tsunami

Tsunami [29] (also known as Kaiten) is one of the earliest IoT bots, but its variants are still being deployed today. It is a DDoS capable bot using IRC as its C&C communication protocol. It is written in C. Its source code is publicly available online, which supports the development of new variants. They all share the main functionality (DDoS and the way of contacting the C&C) [28]. Tsunami variants include Capsaicin (which adds new DDoS methods) or Amnesia (which adds sandbox detection capabilities).

2.1.2 Gafgyt

Gafgyt [29] (also known as Qbot or Bashlite) is an early DDoS capable bot. It communicates over a custom text-based C&C protocol. Gafgyt initially connects to one of the hard-coded IP addresses and announces itself to the C&C. Then, it waits for incoming commands and executes them when received. Gafgyt is also written in C. Variants differ in available DDoS methods and other minor features such as killing other competitor bots (if the target device is already infected). Succeeding variants started reusing parts of Mirai's source code for IP address generation during the scanning phase [28].

2.1.3 Mirai

Mirai is known for its massive DDoS attacks in 2016, reaching 1 Tbps in volume [30]. At that time, Mirai botnet had a steady population of up to 300 000 infected devices, consisting of DVRs, IP cameras, or routers [31]. After several highly effective attacks, the Mirai author publicly shared the source code on `hackforums[.]net`. Since then, it became the most influential IoT botnet, with new variants appearing frequently. Mirai's source code can be divided into three parts: bot, loader, and the C&C [31].

The bot part is written in C, providing capabilities to scan Telnet, communicate with the C&C, and execute DDoS. The scanning module operates via raw sockets. Connections are put in a designated connection table limited to 128 entries by default, storing only the ones with valid SYN+ACK replies. Interestingly, the scanning can be relatively easily detected because Mirai sets the initial TCP sequence number equal to the scanned IP address instead of randomly generating it [32].

Mirai's range of scanned (pseudo-randomly generated) IP addresses is limited by several blacklisted subnets (such as the U.S. Department of Defense). After finding an open Telnet port, it tries to log in 10 times with randomly chosen preconfigured credentials. On success, it reports IP address and credentials to the report server. This information is then used by a separate load module that infects the devices [31]. When Mirai is connected to the target device, it sends a specific command sequence to verify if it has access to a valid Linux shell [32]:

```
enable
system
shell
sh
/bin/busybox MIRAI
```

Mirai's C&C module is written in Golang. It opens two TCP ports for communication with bots, admin, and operators. For bot communication, it implements a simple binary protocol. New bots contact the C&C by sending four bytes of data indicating ID string. It expects the bot to reach the C&C again within 180 seconds. Otherwise, it closes the active TCP connection. Bots regularly send two bytes to the C&C so it can keep track of all active bots. The same port is also used by the admin, who is presented with a login screen when initiating connecting with anything other than four bytes. The second port serves as an API for sending attacks. Users can specify the number of bots to execute the attack, the attack's type and duration, type-related arguments, and target (IP address, list of IP addresses, or an IP range) [33].

2.1.4 Hajime

Hajime [34, 35] was discovered in the same year as Mirai by the Security Research Group at Rapidity Networks. Compared to Mirai, it is more sophisticated. Rather than using centralized C&C as previously described families, it builds a peer-to-peer network to control and monitor its bots. Surprisingly, Hajime has not executed any DDoS attacks, and it was speculated that Hajime was actually deployed with no malicious intention by a whitehat hacker [35]. Right after infection of the device, it blocks commonly exploited ports so that no other malware can infect the machine. Hajime has been distributed in two modules: a peer-to-peer module and an attack module.

The peer-to-peer module adopts Distributed Hash Table (DHT) and µTorrent Transport Protocol (uTP) protocols used in BitTorrent. Hajime contains hard-coded bootstrap nodes in the initial config file, where it can get information about other peers. Further, it searches regularly for new files to download. In DHT, files are denoted by a key (for example, `info_hash` – a hash of torrent's metadata in BitTorrent). Hajime looks up its files by

concatenating the current date with a hexadecimal representation of SHA-1 of the file name, again hashing the concatenated version with SHA-1 [34]. Downloaded configuration and payload files are stored in a custom file format. Also, Hajime encrypts its uTP communication with the RC4 stream cipher.

The scanning routine is similar to Gafgyt [34]. It spreads by randomly scanning IP addresses, excluding particular prefixes on the internet. At the beginning of its development, it was scanning only Telnet with a small dictionary of default credentials. Upon successful connection, Hajime sends a command sequence similar to Mirai [34] to verify access to the Linux shell. The scanning module was later being extended (remember that Hajime had an opportunity to update individual parts of the botnet through the downloaded configuration) to exploit CVE-2015-4464, CVE-2016-10372, CVE-2018-10561, and CVE-10562.

2.1.5 Hide and Seek

In 2018, researchers from Bitdefender discovered another peer-to-peer botnet called Hide and Seek [36, 37]. With its worm-like behavior, it was reported to account for more than 30 000 infected devices [38], although now the botnet does not seem to be active. It implements a custom peer-to-peer protocol. Again, the bot can be divided into a peer-to-peer module and an attack module.

Upon start, Hide and Seek either opens a random port or the one specified via command-line arguments. The bot uses UDP to communicate a predefined set of messages (see Table 2.1 for a summary of the supported messages). The originally described binary came with 14 hard-coded IP addresses (12 of them located in South Korea) as a predefined set of initial peers. New peers are discovered by sending a peer request (`~` message). Target will randomly pick a peer from its peer list and reply with a peer response (`^` message). The bot acquires information about configs by sending a config version request (`h` message) to other peers. After receiving a reply (`H` message) with a larger version value, it queries its peers for new data (`m` and `y` messages). Then, it receives the requested data (`Y` message), verifies it with ECDSA, and adds the data into the config cache (in case the requested file is a new config); or saves the data to a disk and executes it (in case the requested file is a new executable).

The scanning routine is similar to Mirai in that manner that it asynchronously sends SYN packets via a raw socket (originally scanning ports 23, 2323, 9527, 80, and 8080) [36]. Interestingly, it implements multiple approaches to download payload to the infected machine. Firstly, it checks if commands `base64` and `echo -e` are valid on the scanned target system, and in such case, the payload can be delivered as a base64 encoded string. Secondly, it runs HTTP and TFTP servers to host the binaries by itself.

command	description
`h`	config version request
`H`	config version reply
`m`	data request
`y`	data chunk request
`Y`	data chunk reply
`~`	peer request
`^`	peer response
`0`	acknowledgement
`z`	report vulnerable device

Table 2.1: Summary of commands supported by Hide and Seek's custom peer-to-peer protocol [36, 37].

Later in 2018, Hide and Seek was getting updated with newly discovered exploits, a persistence mechanism (using `/etc/init.d` or `/etc/rc.d`). In addition, among one of the distributed executables was cryptocurrency miner [37] (based on the open-source project cpuminer-opt), mining Monero.

2.1.6 Torii

Although not as prevalent as other botnets, Torii demonstrates how IoT malware can further evolve and what new techniques it can adopt. In 2018, researchers from Threat Intelligence Team at Avast described a newly arising botnet [39]. Torii's victims are infected with a shell script that tries to discover the device's architecture and downloads the corresponding payloads via HTTP or FTP. Torii has two stages. The first stage is a dropper, and its sole purpose is to install another executable (the second stage) payload. Torii tries to hide on the system by pseudo-randomly generating its location and filename on the system. Next, it establishes persistence in six different ways and executes its second stage.

The second stage has a hard-coded set of C&C domains operating on TCP port 443. Despite the well-known port, it is not HTTPS traffic but a custom protocol with multiple encryption layers. Communicated messages are first encrypted with a simple XOR-based algorithm. Then, multiple messages are formed into an "envelope" and encrypted with AES-128 with an additional MD5 checksum to ensure message integrity [39]. Options controlled in the commands sent by the C&C server are: downloading and storing new files, executing shell commands, exfiltrating files to the target system, deleting specified files, and updating the C&C address.

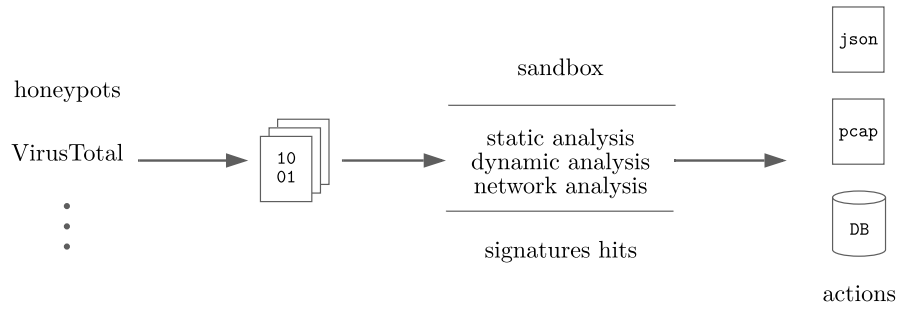

Figure 2.1: Typical automated malware analysis pipeline. The left side of the figure shows incoming malware samples from various sources into a file queue processed by a sandbox system.

2.1.7 Other Variants

Malware researchers frequently report new malware variants that either update or extend the functionality of their predecessors. To comprehend the concept of evolution and extensions of malware families, we can mention a few of the variants. For instance, Fbot [40] with similar architecture to Mirai, yet updating encryption mechanisms; Tor variants of Gafgyt [41] hiding its C&C communication behind over 100 built-in Tor proxies; or Mozi that used DHT in the same way as we have seen for Hajime, but alternatively incorporates new vulnerabilities to scan, and instructions to conduct DDoS attacks, download payloads, and execute commands.

2.2 Automated Malware Analysis

In the previous section, we talked about several IoT malware families. Analysis of actual malware samples is undoubtedly a time-consuming task. When possible, we intent to automate malware analysis tasks to keep up with the number of incoming malware samples. Automated malware analysis pipelines incorporate both static and dynamic processing of malware samples, providing binary, system, and network artifacts in various formats. Figure 2.1 describes generic parts of the pipeline.

2.2.1 Acquiring Malware Samples

The prerequisite for IoT malware analysis is acquiring actual malware samples. This is usually achieved by deploying honeypots or by processing various threat intelligence feeds. Honeypots [42] are systems disguising themselves as real production systems (in our case, IoT devices), serving either as a layer improving a company's security (to lure adversaries) or as a research tool. Different honeypots offer us various levels of interaction with adversaries.

```
import "vt"

rule new_malicious_elf_cz {
  condition:
    vt.metadata.new_file and
    vt.metadata.filetype == vt.FileType.ELF and
    vt.metadata.submitter.country == "CZ" and
    vt.metadata.analysis_stats.malicious > 10
}
```

Figure 2.2: An example of VirusTotal Livehunt rule. Triggering results for new ELF files submitted from the Czech Republic with at least ten detections from different antivirus scanners.

The usual way to classify honeypots is into three categories: low-interaction, medium-interaction, and high-interaction honeypots. One example of an actively used honeypot project is Cowrie[8]. Cowrie is a medium-interaction to a high-interaction honeypot. It emulates SSH and Telnet servers, providing the adversary with an interactive shell and a simple filesystem. The whole project is profoundly configurable: (1) the presented (fake) filesystem can be altered to resemble the actual firmware of a particular device; (2) pre-login and post-login banners (which may be used to identify the target operating system) can be modified; (3) users can implement their own shell commands (stdout responses) available to the adversary; (4) it is possible to predefine accepted Telnet and SSH credentials.

From a threat intelligence feed standpoint, we must mention a popular threat intelligence service VirusTotal[9] which as its main component offers an interface to scan potentially malicious files and URLs using more than 70 antivirus scanners. This way, it has extensive input of new malware samples and allows its users to search ("live hunt") for specific files by creating YARA[10] rules. Using their `vt` YARA module, it is possible to incorporate VirusTotal metadata such as detection names, number of detections, or country of origin. An example of the VirusTotal hunting rule is shown in Figure 2.2.

Finally, there is an option to track peer-to-peer botnets. As was the case for Hajime, Hide and Seek, and Mozi, researchers were able to understand the underlying peer-to-peer communication, allowing them to prepare botnet trackers. A tracker, in this context, is a program that disguises itself as a peer, a valid infected device joining the network. It then parses captured botnet commands, configurations, or newly dropped malicious payloads.

[8] https://github.com/cowrie/cowrie
[9] https://www.virustotal.com
[10] https://virustotal.github.io/yara

2.2.2 Sandbox Analysis

Sandbox is a safe environment to run programs, usually built on top of a virtualization platform and accompanied by additional analysis tools. LiSa (Linux Sandbox) [10] is a multiplatform sandbox system targeting IoT malware. It supplies automated static analysis, dynamic analysis, and network analysis of ELF files. LiSa sandbox's core consists of minimal Linux images, which are emulated on five different processor architectures. A malware sample is executed inside the Linux image while tracing system calls and recording network traffic.

From the traced system call activity, we can reconstruct the executable's interaction with the system, giving us an overview of executed commands, the number of processes, data sent to sockets, or files that have been manipulated (opened, created, or deleted).

Network traffic is recorded in pcap format, which can then be analyzed by any packet analyzer software (such as Wireshark). Besides, LiSa by itself imparts fundamental analysis of pcap files. It outputs statistics about accessed ports, number of initiated connections with TCP SYN packets; it parses several application-level protocols to reconstruct DNS queries, HTTP requests, IRC messages, and Telnet data [10].

Both static artifacts and behavioral artifacts can be matched with YARA rules. Well-written YARA rules can serve as a description of a specific malware family. Additionally, we gain visibility inside the types of executables we are running and the possibility to label all the output data appropriately.

2.3 Network Behavior

We reviewed what IoT malware families we know and how to get information about their behavior in an automated manner. This section summarizes our knowledge about the networking aspect of malware's behavior. Consequently, we can proceed with the design of specific detection mechanisms of the said behavior in further chapters. From a higher-level perspective, a sequence diagram of typical botnet behavior is illustrated in Figure 2.3. We identified the major distinct categories we needed to focus on: the C&C communication and any other control methods, scanning, infection, DDoS, and cryptocurrency mining.

2.3.1 C&C Communication

Centralized C&C communication typically uses TCP as its underlying transport protocol. Bots establish a connection with the C&C server, and this connection can be many hours or days long. Bots tend to report themselves periodically via so-called heartbeat messages. Note that heartbeat messages do not have reason to change over time, so an isolated view of the messages will

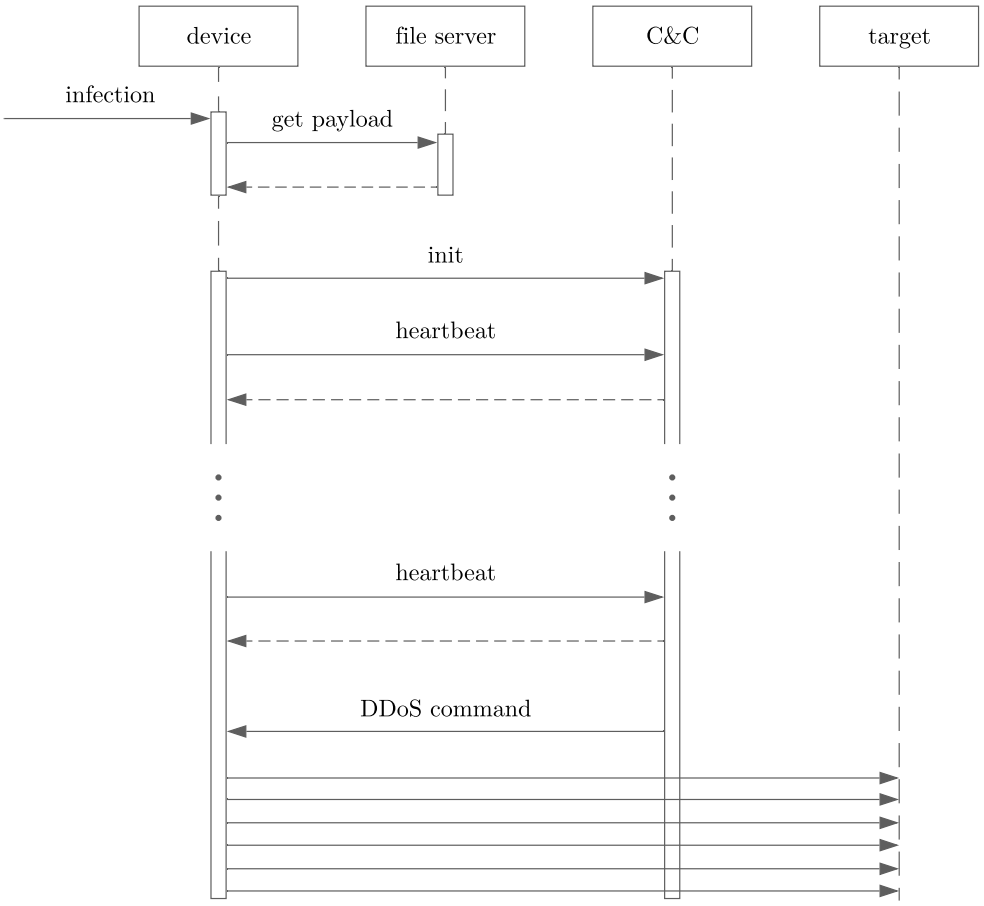

Figure 2.3: Sequence diagram of a typical (in the sense that almost 90 % of bots are representatives of etheir Gafgyt, Mirai, or Tsunami families) botnet communication [28]. For clarity, scanning of random endpoints on the internet is not illustrated.

show low-entropy data with significant directional inter-arrival times. Other than a heartbeat, bots C&C communication channel transmits bot commands. The commands can trigger other actions such as scanning or DDoS. Commands' syntax differs significantly between different malware families and their variants. Although it is possible to detect bot commands applying signatures (and many such signatures exist in the ET OPEN ruleset), it is not a scalable solution because only minimal program changes are required to avoid detection.

2.3.2 Scanning and Infection

Internet scanning by itself is a broad subject. Public community is usually familiar with two popular projects for port scanning - nmap[11], and masscan[12]. Using these scanning programs, attackers can quickly get informed about potentially vulnerable devices with open targeted ports in the specified subnets. Afterward, they will give this information to a custom exploiting module that tries to infect the device. To merge these two operations into one, attackers often develop their own scanning modules. They can be implemented as synchronous or asynchronous routines; they might try to scan the whole internet or try to be stealthy by slow scanning. The routines can be implemented as part of the bot (introducing worm-like features), or attackers can run separate scanning modules from their own ad hoc infrastructure.

After the scanning phase, there is an infection phase. The infection effort shall be represented with many login attempts to Telnet and SSH servers. Bots typically have predefined dictionaries of default passwords, which can be adjusted to fit a particular device. Another popular attack vector is via remote code execution (RCE) vulnerabilities. Malware authors can monitor newly discovered vulnerabilities in exploit databases[13] and adopt the presented proof of concept exploits into their code. Mainstream botnets, independently of the family, try to implement a similar (as large as possible) set of exploits to enlarge the attack vector and further expand their botnet. To reflect the current state of the exploits, we can take a look at the vulnerabilities exploited by Mozi botnet [43] (see Table 2.2).

2.3.3 DDoS

Bots start attacking right after receiving the command from the C&C, effectively disrupting the service's availability if the botnet is large enough and the target does not have any DDoS mitigation solution in place. Several types of DDoS attacks are there to choose from, and the bots generally implement multiple distinct techniques. A flooding attack is the simplest example of a DDoS attack. Its goal is to occupy the target's bandwidth so that no legitimate user can connect to the provided services. Prevalent types of flooding attacks are UDP flood and ICMP flood because it is practicable to choose connectionless protocols. Amplification attacks are another method for DDoS attacks, leveraging a request-response type of communication, where responses are ideally much larger than the original request. In this case, an attacker sends spoofed requests to a so-called reflector. The reflector sees as a source IP address the DDoS target, sending the response data to it. Lastly, there are types of DDoS attacks exploiting some specific properties of communication proto-

[11] https://github.com/nmap/nmap
[12] https://github.com/robertdavidgraham/masscan
[13] https://exploit-db.com

target	vulnerability type	cve
Vacron NVR	HTTP GET	2008-4873
Realtek SDK	UPnP SOAP	2014-8361
Netgear R7000 and R6400	HTTP GET	2016-6277
Huawei HG532	UPnP SOAP	2017-17215
D-Link (multiple devices)	HNAP SOAPAction	2015-2051
GPON	HTTP GET	2018-10562
CCTV DVR (multiple brands)	HTTP GET	n/a
D-Link (multiple devices)	UPnP SOAP	n/a
Eir D1000	UPnP SOAP	n/a
Netgear DGN1000	HTTP GET	n/a
MVPower DVR	HTTP GET	n/a

Table 2.2: Vulnerabilities exploited by Mozi [43]. All mentioned vulnerabilities are command injections vulnerabilities allowing the attacker to simply run any command (conviniently downloading a malware and executing it).

cols. An example of this type of DDoS attack is a TCP SYN attack which tries to exploit the implementation of a TCP three-way handshake, where the server may exhaust its resources while waiting for ACK packets to finish the connection setup [44].

DDoS methods presented in Mirai and its variants are configurable via attack flags. Botnet operators can specify desired IP or TCP header fields (such as TTL, TCP flags, or sequence numbers) or payload data. Apart from generic UDP, SYN, or ACK floods, it can execute several target-specific attacks, aiming, for instance, at HTTP services by generating many HTTP requests, DNS servers by generating recursive DNS requests with random data, Valve game servers by generating UDP packets on port 27015 with `"TSource Engine Query\x00"` data [33].

2.3.4 Cryptocurrency Mining

Cryptocurrency mining is one of the ways to monetize an established botnet. Undoubtedly, the most popular cryptocurrency within cryptomining malware is Monero (XMR) [45]. Malicious users tend to use Monero for its privacy and suitability for CPU mining. Monero is using Cryptonote as its Proof-of-Work (PoW) algorithm. Cryptonote is designed to be more memory-demanding than Hashcat (Bitcoin's PoW), making mining on CPUs more efficient (and less efficient on GPUs) [45].

We discussed that some botnets have the ability to download new executables and thus extend their functionality. We already mentioned cpuminer-opt and its usage by Hide and Seek. Another example of an open-source miner

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "method": "login",
  "params": {
    "login": "login",
    "pass": "pass",
    "agent": "XMRig/6.4.0 (Linux x86_64) libuv/1.38.1 gcc/9.3.0",
    "algo": ["rx/0", "cn/2", "cn/r", "cn/fast", "cn/half", ...]
  }
}

{
  "jsonrpc": "2.0",
  "method": "job",
  "params": {
    "blob": "0e0ec4fca5830675bce6dcfc...",
    "job_id": "6423",
    "target": "f3220000",
    "height": 2331805,
    "seed_hash": "51bad5e79425f972406e5826dc21...",
    "next_seed_hash":""
  }
}
```

Figure 2.4: Examples of Stratum messages. The upper message, containing login information, appeared in the first packet of the established connection. The lower message contains a job specification coming from the pool.

that is often downloaded and executed by the bot is xmrig[14].

Rather than mining alone, bots join particular cryptomining pools. In cryptomining pools, devices share their computational power to have as a whole a higher probability to mine a new block and get the reward. Few protocols to distribute work to individual contributors inside the pool exist; however, at least for Monero, we recognize the popularity of the Stratum protocol. Stratum uses TCP as its underlying transport protocol, and as its payload, it sends JSON messages, which are by default sent in cleartext. Figure 2.4 shows two example Stratum messages.

[14] https://github.com/xmrig/xmrig

Chapter 3

Datasets

IoT network security domain lacks a ground truth up-to-date dataset. Preparation of such a dataset introduces many domain-specific problems [46]. Creators of the dataset need to choose the form of the data. Datasets can be captured on the packet level (in the form of pcap), on the flow level, or in some other form of structured data and metadata. Additionally, authors need to deal with privacy if the data is not captured in a simulated or strictly controlled environment. Raw data often contains private information and must be thoroughly anonymized. In the specific case of IoT malware, creators must also have access to malware samples and an environment to run the samples. When collecting malware traffic, the environment tends to be artificial and biased towards malicious data. The perfect dataset [46] would contain both normal and malicious behavior over a long time. Since attacks are rare events, the final dataset would be highly imbalanced [47] and the individual classes would be distributed according to some measured or expected frequency.

3.1 UNSW IoT Traces

In [48], the authors studied the possibilities of machine learning methods concerning IoT device classification tasks. As part of the research, they released a dataset containing traffic (named by the authors as traffic traces) of 28 unique devices. The devices are divided into six categories: (1) cameras, (2) controllers and hubs, (3) energy management devices, (4) appliances, (5) healthcare devices, and (6) non-IoT devices such as laptops and smartphones. All the devices were interconnected on a single LAN. Their MAC addresses inherently define the devices' labels that authors used for classification. The traffic was captured using tcpdump, and the data is published in the form of pcaps. Each pcap is 24 hours long, and at the time of writing, the dataset contained 61 separably accessible pcaps. Further in the thesis, we will refer to the said data as UNSW IoT Traces dataset.

We processed the provided pcaps into flows using CESNET ipfixprobe and labeled all flows as benign, as we do not need to discriminate any devices' types. Moreover, we filtered out any communication between two devices on the same LAN. The LAN traffic would not be generally visible by deployed flow exporters; hence, we are strictly interested in communication with the internet, possibly going through metering probes.

3.2 Aposemat IoT-23

Aposemat IoT-23 [49] is a dataset prepared by researchers at Stratosphere Laboratory at CTU. This research group already published another better-known dataset, called CTU-13 [50], which covered several Windows botnets. On the other hand, IoT-23 focuses solely on IoT malware families. The datasets are divided into captures, called scenarios (13 and 23 of them, respectively). Each scenario consists of a pcap capture and other higher-level data. In CTU-13, data is also represented in the form of bidirectional flows. In IoT-23, we are given Zeek flows – specifically, modified versions of Zeek connection files containing assigned labels. The main advantage of both datasets is their length. Because the authors focus on malware's long-term execution, each scenario is, on average, several hours long.

The infected device in all IoT-23 malicious scenarios is a Raspberry Pi. Summary of scenarios mentions several IoT malware families; namely, it is Mirai, Torii, Gafgyt, Kenjiro, Okiru, Hakai, Hajime, Muhstik, and Hide and Seek. The devices in benign scenarios are Amazon Echo, Philips Hue, and Somfy smart lock.

According to the IoT-23 documentation [49], pcap files were manually analyzed by an analyst. The outcome of the said analysis was a set of rules used to automate the labeling process. Final labels are presented in Zeek connection logs. Labels are either generic (attack, benign, C&C, DDoS, FileDownload, Heartbeat, and PartOfAHorizontalPortScan) or family-specific (Mirai, Okiru, and Torii). A significant part of the flows is also labeled with a combination of labels (such as C&C-Torii).

We dug deeper into all the scenarios to validate labeled C&C flows in which we are primarily interested. We found out that the provided labels do not match their explanations. One illustration of label inconsistency is the second biggest scenario, CTU-IoT-Malware-Capture-17. 6834 flows in this scenario are labeled as C&C-HeartBeat. IoT-23 documentation [49] states that the C&C label *"indicates that the infected device was connected to a C&C server"*, and the HearBeat label *"indicates that packets sent on this connection are used to keep track of the infected host by the C&C server"*. However, all labeled HeartBeat flows are all non-initiated connections, produced by 20 684 TCP SYN packets and 374 TCP RST packets.

The authors of the dataset were aware of this situation, as similarly labeled

connections are prevalent in the rest of the scenarios too. In their approach, they considered unsuccessful connections to an inoperative server as valid C&C traffic. However, we preferably want to discard all such connections as they, apart from IP addresses, do not carry any information related to the malware family. Hence, we first filtered the pcaps for potential C&C connections by IP addresses and ports taken from the labeled Zeek connection logs; we extracted successfully initiated connections using tshark; and, finally, we processed the final pcaps with ipfixprobe.

3.3 Custom Dataset

To supplement the existing datasets, we decided to prepare our custom C&C dataset. In the following sections, we describe its preparation procedure, so it is reproducible even for new malware families and can serve as a solid baseline for upcoming experiments.

3.3.1 Environment Setup

The backbone of our virtualized environment is libvirt[15], a robust virtualization API. Utilizing libvirt, we can easily create a virtual network for our guest machines. In our situation, we need to prepare two guests – a C&C server and a victim device. For the C&C server, we used Debian, but any Linux distribution having the prerequisite packages needed to run the C&C software would be suitable. For the victim device, we decided to reuse the guest virtual machines of the LiSa sandbox.

Setting up Debian under libvirt is a rather straightforward task. Following the tutorial on Debian KVM wiki pages[16], we set up XML configuration for the C&C guest image. The difference with LiSa's images is that we also need to incorporate a separately compiled kernel and filesystem. To do this, we define the `os` XML section:

```
<os>
  <type arch='x86_64' machine='pc'>hvm</type>
  <kernel>/lab/x86_64/images/bzImage</kernel>
  <cmdline>root=/dev/vda</cmdline>
  <boot dev='hd'/>
</os>
```

and the `disk` section:

[15] `https://libvirt.org/`
[16] `https://wiki.debian.org/KVM`

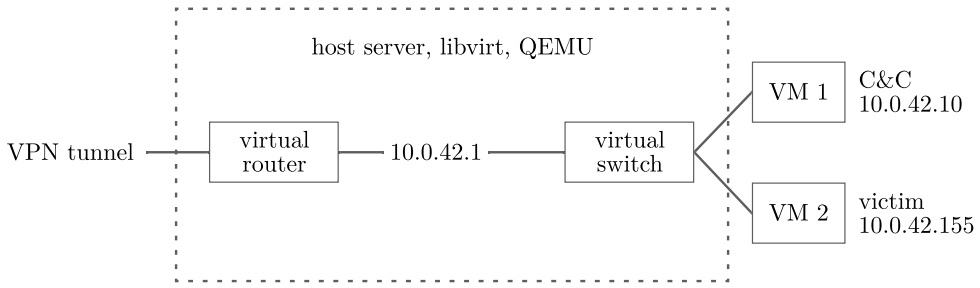

Figure 3.1: Virtual lab environment network diagram.

```
<disk type='file' device='disk'>
  <driver name='qemu' type='raw'/>
  <source file='/lab/x86_64/images/rootfs.ext3'/>
  <target dev='vda' bus='virtio'/>
</disk>
```

Configured guest machines are then connected within `10.0.42.0/24` network, with all traffic sourced from this IP range being routed via VPN (see network diagram in Figure 3.1).

3.3.2 Source Code Acquisition

Now, as we have the environment ready in place, the simplest way to fully control the malware is to deploy it ourselves. Compared to running malware samples captured in honeypots or downloaded from VirusTotal, compiling bots ourselves gives us the power to simulate botnet activity without any harmful actions. Besides, this way, we can quickly produce valid labels, annotating bot commands, as we were the ones sending them.

We started by searching for leaked malware source codes. IoT botnet threat landscape is ruled by script kiddies who tend to brag about their accomplishments publicly on the internet, so it does not take much time, and one may find tutorials for setting up botnets, forum posts, videos, or Instagram accounts dedicated to selling the botnet usage as a service.

We joined a few of their communities on Discord, started scraping channels for attachments, and downloaded 70 different malware source codes. They were all modifications of Gafgyt and Mirai. All released under different names, advertised in the community as different "sources". At some point in time, most of these sources were private modifications sold to other botnet operators, and they got leaked eventually after the source code was sold to enough people. Modifications done in the acquired versions are mainly new scanning methods to leverage publicly known exploits.

A similar database of leaked botnet source codes was until recently available at Darknetleaks archive[17]. Fortunately, we have downloaded everything present in the archive before its shutdown. Apart from other tools, scanners, and some database dumps, it hosted 180 malware sources. Approximately the same content has also been released by the Threatland group (whose Twitter account got suspended soon after the release) on Github[18].

3.3.3 Botnet Deployment

Since we found out that C&C communication is not the subject of modifications between versions, we did not have to deploy several versions of the same botnet family. We picked one source from each category (1) Tsunami, (2) Gafgyt, (3) Mirai. Each of those implements a distinct communication protocol; Tsunami is an example of an IRC bot; Gafgyt uses a simple text-based protocol; Mirai implements a custom binary protocol.

Tsunami

To control IRC bots, we must set up an IRC server. Many tutorials distributed alongside Tsunami variants encourage using UnrealIRCd[19], and they also deliver its configuration files. Other sources distribute only one file (implementation of the bot part) as it is evident that any IRC server would work. After configuring and starting our IRC server, we must modify the bot source code to point it there. We changed hard-coded definitions of the destination port, channel, and server's domain name within one file. Then, we just compiled the bot for the target architecture and transferred it onto the victim device.

Gafgyt

Gafgyt's `server.c` was compiled as-is. Additionally, we prepared `login.txt` file to permit our operator to login and control the botnet. `client.c` must have a hard-coded IP address of our server, so we changed that. We again transferred the compiled bot onto the victim device and started the server providing its arguments (bot port, number of threads, and C&C port).

Mirai

Original post at Hack Forums[20] included an elaborate tutorial on how to set up the bot. We followed the tutorial and configured C&C domain and port in `bot/table.c`, which holds xor encoded entries of numerous values used

[17] `darknetleaks[.]ru`
[18] `https://github.com/threatland/TL-BOTS`
[19] `https://www.unrealircd.org/`
[20] `hackforums[.]net`

throughout the program. We can use the included encoding tool to generate table entries, but it is straightforward to do it by ourselves. Taking a closer look at the source code, we see that both `tools/enc.c` and `bot/table.c` contain a routine that uses hard-coded four bytes of `table_key`. The routine iterates over each byte of our value and applies XOR for each byte of the `table_key`. We are unsure if the author was hoping for some false sense of security, but it is obvious that this routine can be simplified to an algorithm with a key of the length of only one byte. Afterward, we set up a MySQL database and created the necessary tables. Finally, we edited the correct database details in `cnc/main.go` and we were ready to go.

3.3.4 Network Capture of Different Scenarios

We decided to start the malware in a controlled manner, filtering out its scanning and exploiting capabilities. We were capturing the traffic in pcap format using the tcpdump tool during all experiments, later processing them with ipfixprobe to extract the necessary flow records.

Our dataset should cover most of the notable C&C behavior. We previously recognized C&C communication as C&C heartbeat and bot commands. Thus, for each of the three prepared malware families, we define two scenarios to capture both C&C communication components. Each scenario should have multiple representatives to express its generic properties. We define a representative unit as packets belonging to the TCP connection with the C&C server within a five-minute time bin. Five minutes were selected because it is a common value for active timeout inside NEMEA ipfixprobe exporters.

In the first scenario, we imagine the malware running in the background with no received commands. This scenario includes the initiation of the TCP connection to the C&C server, which is then continuing for one hour. Thus, producing 12 representative groups of packets belonging to this scenario for each malware family.

In the second scenario, we imagine the malware receiving commands from its C&C server. We produced a representative group per command, per malware family. Representative groups of the second scenario consist of packets associated with the current command and heartbeat packets in the background. The position of command packets among background heartbeat packets is chosen arbitrarily because, in the real-world scenario, the timing of the commands is tied to a random human action. Each family supports a different number of commands. Gafgyt variant we deployed supporting four commands, Tsunami and Mirai variants supporting both eight commands. Thus, in total, this scenario has 20 representative groups.

Chapter 4

Design

Previous chapters showed that there are many aspects of both network infrastructure monitoring and network behavior of IoT malware that must be taken into account when designing any detection system. In this chapter, we first review the design objectives that arose from the analysis of network monitoring principles, applicable not only to the NEMEA system. We then introduce the basic concepts of our design. We address the most important malware behavioral aspects individually in separate sections. Finally, we propose a method to detect infected IoT devices in a computer network using combinations of network indicators.

4.1 Objectives

Motivated by the discussed characteristics of network monitoring and IoT malware, we state that the designed system should fulfill the following functional requirements:

- It *must* be implemented as a flow-based solution, not relying upon extensive packet capture, processing (extended) IP bidirectional flows. The gravity of IP flow extension shall be defined by the available NEMEA ipfixprobe plugins.

- It *must* be tolerant to nuances caused by high-speed network monitoring infrastructure states. Both early export and direction mismatch must be considered. Early export causes flows to be exported to the collector before the configured active timeout because of resource exhaustion or flow key hash collision. Thus, receiving less data in the exported flow. Direction mismatch happens because the direction of bidirectional flow is decided based on the first seen packet, disrupting our sense of which endpoint initiated the communication.

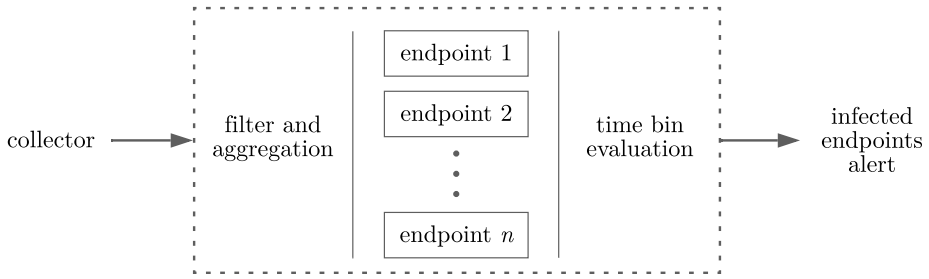

Figure 4.1: Top-level design. The proposed detection system receives collected and aggregated flows. Unrelevant flows are filtered out to target only specific monitored endpoints. Per device evaluation then happens for each time bin. The system identifies infected devices and provides indicators to justify its decision.

- It *should* be scalable enough to cope with a possibly growing number of devices in the network.
- It *should* be appliable to both home network security monitoring and ISP-level monitoring. Underlying principles must not depend on LAN visibility.
- It *should* present interpretable results. Operators should distinguish which aspect of the device's behavior caused the alert. They should also be able to understand the detection mechanism, so they are eventually able to closely examine false positives.
- It *could* use statistical information of extended IP flows so that the designed system introduces future-proof concepts when more and more network traffic is going to be encrypted.

4.2 Conceptual Design

Figure 4.1 shows a top-level view diagram of our proposed design, compatible with the stated functional requirements. The detection system is preceded by a collector and an aggregation module, possibly receiving flows from multiple flow exporters. We shall time aggregate flows to a uniform length to minimize the effects of timeouts and flow cache hits. The sole system then consists of flow filtering and a structure of detection modules that keep individual devices' detection results. Results are evaluated per time bin of fixed configurable length. The system thus provides a periodic output for each monitored endpoint. Positive output (device's infection) is also annotated with a set of indicators.

The structure of detection modules intends to target different malware's network behavior separately. This way, it is possible to choose an appropriate technique and dataset for the specific problem. Although detection modules do not have to build on top of any machine learning nor statistical methods, we will adopt the terminology used in the machine learning field. Further, we denote each detection module as a classifier in the form of $\hat{y} = f(x)$; $\hat{y}$ being the predicted class, and x being a classifier-specific feature vector derived from processed flows. We will stick to only two classes, thus solving the binary classification problem. In our design, we combine the output of different classifiers to make the final prediction.

4.3 C&C Communication

Regularity and low entropy of heartbeat messages, short bot commands, and the fact that the C&C channel appeared to send only a few data over time motivated us to look at statistical approaches to distinguish C&C communication apart from normal behavior. To assess our assumption that this communication can be indeed identified based on statistical information, we derived a set of candidate features, analyzed C&C and benign traffic, and prepared and evaluated a baseline model.

4.3.1 Feature Engineering

We derived a set of statistical features for the initial experiments. Derived features were influenced by the features used for VPN and Tor detection by Draper et al. [23, 24]. A list of the used features and their categorization can be seen in Figure 4.2. Computed features are divided into five categories:

1. *Basic traffic statistics* represent the amount of traffic in bytes and packets belonging to a flow. Statistics are derived separately for the forward direction, the backward direction, and the sum of both directions. All statistics are normalized by the flow's duration, thus referencing per-second values.

2. *TCP flags statistics* provide information about the presence of individual bits in TCP flags bitfield for the first n packets, their absolute counts, and their ratios corresponding to all other appearing flags.

3. *Packet length statistics* express the sizes of the first n packets, producing minimum, maximum, mean, and standard deviation for the complete set of n packets and its forward and backward subsets (if any packets of each direction happened to be in the first n packets).

4. *Inter-arrival times statistics* are derived from the timestamps of the first n packets (and their subsets in the same way as for packet lengths

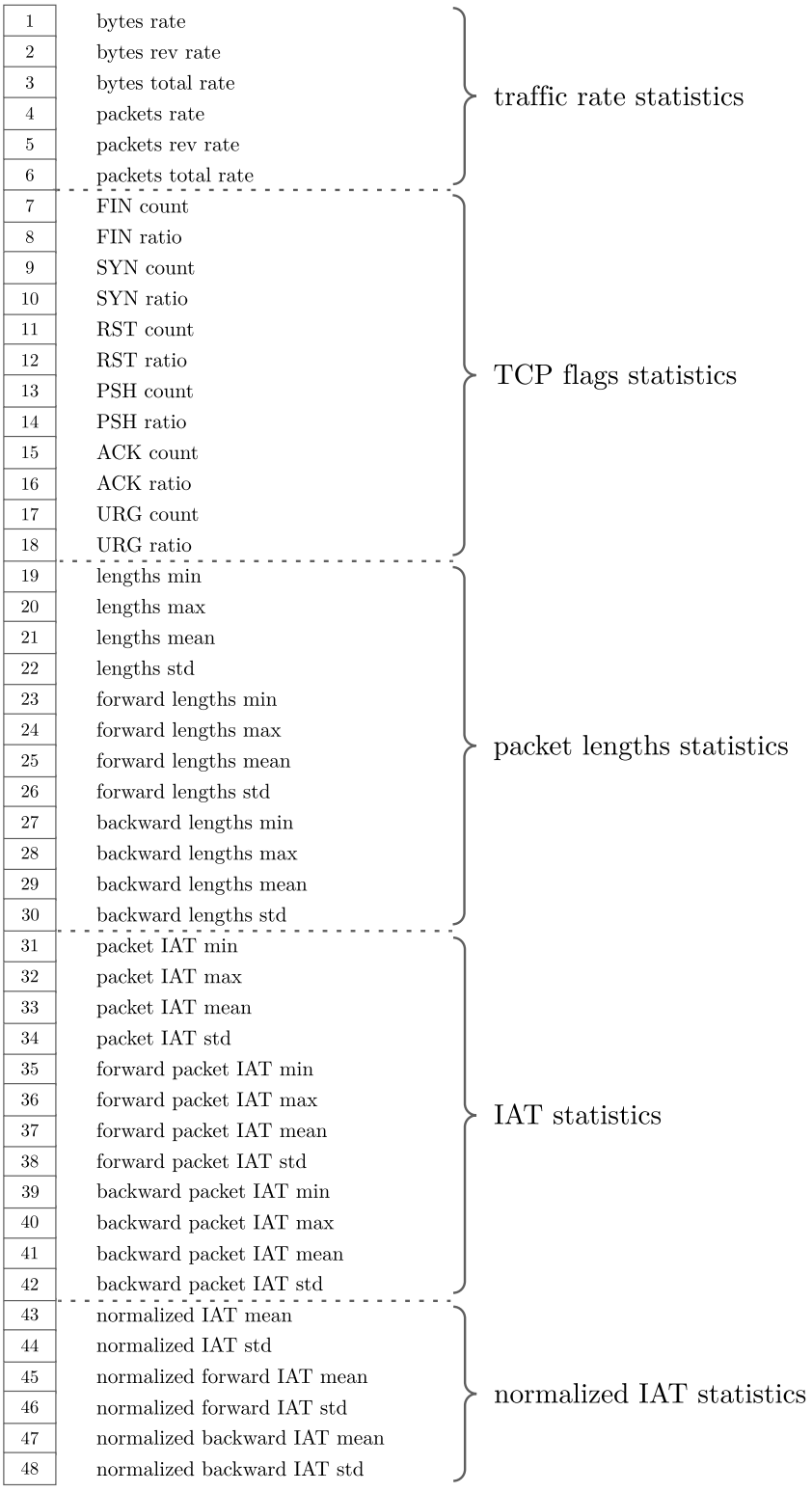

Figure 4.2: List of derived features used for C&C classification experiments.

statistics). Consecutive packets' timestamps are subtracted, giving us inter-arrival times. For forward and backward statistics, we account only for consecutive packets in the stated direction. Again, we derive the minimum, maximum, mean, and standard deviation of these values.

5. *Normalized inter-arrival times statistics* differ only in the way of representing inter-arrival times. Instead of calculating exact inter-arrival times between consecutive packets' timestamps, we divide them into two categories: short and long. Category membership is determined by comparison with a fixed threshold of five seconds.

Extracted features were analyzed on the instances from our custom C&C dataset, merged with benign instances from the UNSW IoT Traces dataset. During data preparation, we first swapped all directions so that the monitored endpoints act as sources of communication. In all previously analyzed cases of C&C communication, it was initiated by the bot. This way, we also do not have to worry about flows' direction mismatch. In our data, we noticed a few highly correlated features. Particularly, backward bytes rate, bytes total rate, forward packets rate, and backward packets rate were all more than 99 % correlated with packets total rate. Thus, for the model's training, the only features kept from the basic traffic statistics were packets total rate and forward bytes rate. In addition, the maximum of all inter-arrival times was in our case 99.9 % correlated with the forward inter-arrival times, and thanks to our way of viewing the traffic with the bot as the source, we kept only forward inter-arrival times.

4.3.2 Model Selection

Our primary concern for model selection is the model's interpretability. Although nowadays, there are many popular machine learning techniques based on neural networks, they act rather like black-box models to their end users. End users can hardly follow thousands or even millions of mathematical operations and the corresponding learned weights used for predictions [51]. On the other hand, the models we considered had either a simple mathematical formulation or other means of extracting the prediction causes, which could be followed in the case of unexpected behavior. Below, we summarize the models we chose for experiments and evaluation.

Logistic Regression

Logistic regression is a linear model for binary classification tasks, also extendable to multiclass classification through a transformation using, for instance, the one-vs-rest method. As might be expected, similarly to linear models for regression, it performs well if the data is linearly separable. In

the logistic regression model, we first represent our data as weighted sums of its features [51, 52]:

$$\mathbf{w}^T\mathbf{x} = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n \tag{4.1}$$

Consequently, we model the probability of a class by taking the weighted sum as an argument of the logistic sigmoid function:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{4.2}$$

Note that the sigmoid function is a monotone function with $\lim_{z \to -\infty} \phi(z) = 0$ and $\lim_{z \to +\infty} \phi(z) = 1$. Therefore, the probability of target variable y being equal to class 1 can be written as

$$P\left(y = 1|\mathbf{x}, \mathbf{w}\right) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}} \tag{4.3}$$

We compute the weights $\mathbf{w}$ by maximizing the log-likelihood function for the training dataset, using optimization algorithms such as gradient ascent [52].

K-Nearest Neighbors

K-nearest neighbors (KNN) is a simple instance-based machine learning algorithm. Instance-based learning algorithms memorize the training dataset. In the case of KNN, the training dataset is used for predicting target values of new instances by finding the closest training instances (neighbors) based on the chosen distance metric. The predicted class is later determined by the majority vote of these k neighbors [53, 52].

The choice of the distance metric depends on our instances' domain. If our instances are vectors of real numbers, it is common to use Mankowski distances, resulting, for example, in the Manhattan distance for $p = 1$ or the Euclidean distance for $p = 2$ [52]:

$$d\left(x^{(i)}, x^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p} \tag{4.4}$$

For the classification of network flows, KNN leads to a natural way of comparing observed flows with previously captured, already known, flows. End users can interpret the classification results if they are familiar enough with the training dataset since they can be presented with the actual list of an instances' neighbors.

Two properties of KNN may discourage us from using it. Firstly, with a large enough dataset, the predictions get computationally expensive, considering we have to find neighbors for every prediction. However, we can consider using some of the existing algorithms for approximate searches, such

as kd-tree, that try to eliminate this complexity [53]. Secondly, KNN is heavily affected by the so-called "curse of dimensionality". With more features (dimensions), the feature space becomes more sparse. In theory, to keep up with the number of features, we would have to increase the size of our training dataset exponentially. Techniques for dimensionality reduction like PCA, which address this issue, are not viable in our case because the original semantics of the computed features would be lost [53, 52].

Decision Tree

Decision trees pose a very intuitive way to predict values. We make predictions by asking a series of questions represented as a tree, going from the tree's root down to the leaf nodes, which carry the final classification labels. In practice, binary decision trees are used to avoid combinatorial explosion with a growing depth of the tree. Splitting decisions are made on one feature at a time. At each split, we can either evaluate if a categorical feature is equal to the splitting value or compare a real number feature to a splitting threshold [53, 52].

Because the construction of the optimal binary decision tree is an NP-complete problem [54], in reality, decision tree learning algorithms work heuristically. Generally, we start with all our training data in a single node, recursively finding the best splits according to some improvement measures [53]. Below, we denote two commonly used measures: entropy (I_H), and Gini impurity (I_G) [52]:

$$I_H(t) = -\sum_{i=1}^{c} p_i \log_2 p_i \tag{4.5}$$

$$I_G(t) = 1 - \sum_{i=1}^{c} p_i^2 \tag{4.6}$$

We also need to keep in mind the overfitting of the model. Decision tree construction may lead to deep trees that perform well on the training set but perform poorly on unseen data. This problem can be addressed by pruning the constructed tree to reduce its depth or, more commonly, by having hyperparameters of the model limiting the splitting procedure (such as the maximum depth of the tree) [53].

Although simple decision trees do not tend to be the best performing classifiers, their interpretability poses a significant advantage. We are able to compose individual decision rules by proceeding from the root node, connecting the splitting conditions with the conjunction operator, down to the leaf node. We can also determine the feature importances and see which of our features affect the prediction the most. Nevertheless, interpretability may get more complicated if the decision tree is too deep and decision rules consider many features [51].

Random Forest and AdaBoost

Bagging and boosting are two conventional methods to create an ensemble of multiple classifiers, random forest representing bagging algorithms and AdaBoost representing boosting algorithms. Random forest builds a collection of n decision trees. Each decision tree is trained on a subsample of the training dataset taken randomly and with replacement. Besides, for each node, we select only d features to consider for the split. The final prediction of a random forest is then a majority vote of all n decision trees [53, 52].

AdaBoost is an algorithm focusing on leveraging so-called weak learners. Weak learners overtake easy-to-learn information respecting the problem, often with only slightly better performance than random guessing. Shallow decision trees (such as decision stumps, decision trees with a depth equal to one) are typical examples of weak learners. AdaBoost proposes that a linear combination of differently focused weak learners can result in a well-performing classifier. This is achieved using an iterative algorithm while manipulating the distribution of weights for dataset instances. We start with a uniform distribution. At each iteration, one weak learner is trained on the weighted dataset. Then, we recompute the weights, and we put more emphasis on misclassified instances. The maximum number of weak learners we want to generate is specified as a hyperparameter. To get the final prediction, we compute the weighted sum with a coefficients $\alpha_t = \frac{1}{2} \ln \left(\frac{1}{\varepsilon_t} - 1 \right)$, inversely proportional to weak learners' weighted error rates ε_t [53, 52].

4.3.3 Baseline Model

To validate our set of features and to conduct experiments giving us immediate results, we started with a baseline model. We chose logistic regression because its simplicity allowed us to have low training times as the model's only hyperparameter we tuned was the regularization constant.

We wanted to explore if the proposed classifier can be trained solely on the C&C instances from our custom dataset described in Section 3.3 so we prepared a training dataset consisting of 54 instances labeled as C&C flows and 5046 instances labeled as benign flows. Benign flows were extracted from the first three days of the UNSW IoT Traces dataset.

Classification tasks of malicious events are typically class-imbalanced problems. Note that C&C instances accounted for about one percent of the training dataset, reflecting the rarity of their occurrences in real data. The basic approach to deal with an imbalanced dataset is to use random undersampling and oversampling, simply randomly choosing instances from the majority class that should be removed and instances from the minority class that should be replicated [55]. For oversampling, instead of replicating existing instances, we used SMOTE algorithm, which generates synthetic instances by finding the

nearest neighbors among the minority class and creating a new instance along a line segment connecting two neighbors in the feature space [55].

The model's pipeline incorporates SMOTE, feature standardization, and logistic regression. As already said, we need to tune only the regularization constant. We set up a routine evaluating multiple values for this hyperparameter. The best hyperparameter was chosen based on the average ROC AUC in k-fold cross-validation. During k-fold cross-validation, we split the training datasets k times (in our case five times), use $k-1$ splits for training, and the last split for model evaluation. This procedure is repeated k times until all splits have taken their round in evaluation part [52].

Thanks to the feature standardization, the absolute values of logistic regression coefficients also represent the feature importances. As we anticipated, packet length statistics and inter-arrival times statistics accounted for the most influential features. Figure 4.3 shows empirical cumulative distribution functions for the top six features.

To estimate the performance of the model, we prepared a testing dataset consisting of 1776 C&C flows extracted from the IoT-23 dataset and 30 422 benign flows extracted both from the IoT-23 dataset and the UNSW IoT Traces dataset. The merge of benign data was due to the low amount of benign flows present in the IoT-23 dataset. We used a total of 12 days of UNSW IoT Traces data (which were all different from the days used during training).

See that the testing dataset is much larger compared to our training dataset. Typically, we would like to use a more significant portion of the data for training. However, we aspired to experiment if the model can perform well even though it was trained on a low amount of data (that could be in the future captured regularly during sandbox analysis). Besides, we would like to show that the model does not depend on the capture environment since the training C&C data was prepared in our virtualized environment, and the testing data was captured by IoT-23 creators on real infected devices. Table 4.1 shows multiple performance metrics for this model (for metrics descriptions, refer to Section 6.1, where we further evaluate other models). The confusion matrix in Figure 4.4 shows that we recorded 52 false positives (0.17 % of all benign instances) and 80 false negatives (4.5 % of all C&C instances).

4.4 Anomaly Detection

After C&C traffic classification, we focused on DDoS and scanning detection. Traffic patterns of scanning and DDoS discourage us from using the similar flow classification approach described in the previous section. For example, the scanning behavior will typically produce many flows, consisting of only one packet. This flow by itself does not provide enough information to tell if it is malicious or not. We propose a method based on time-series anomaly detection

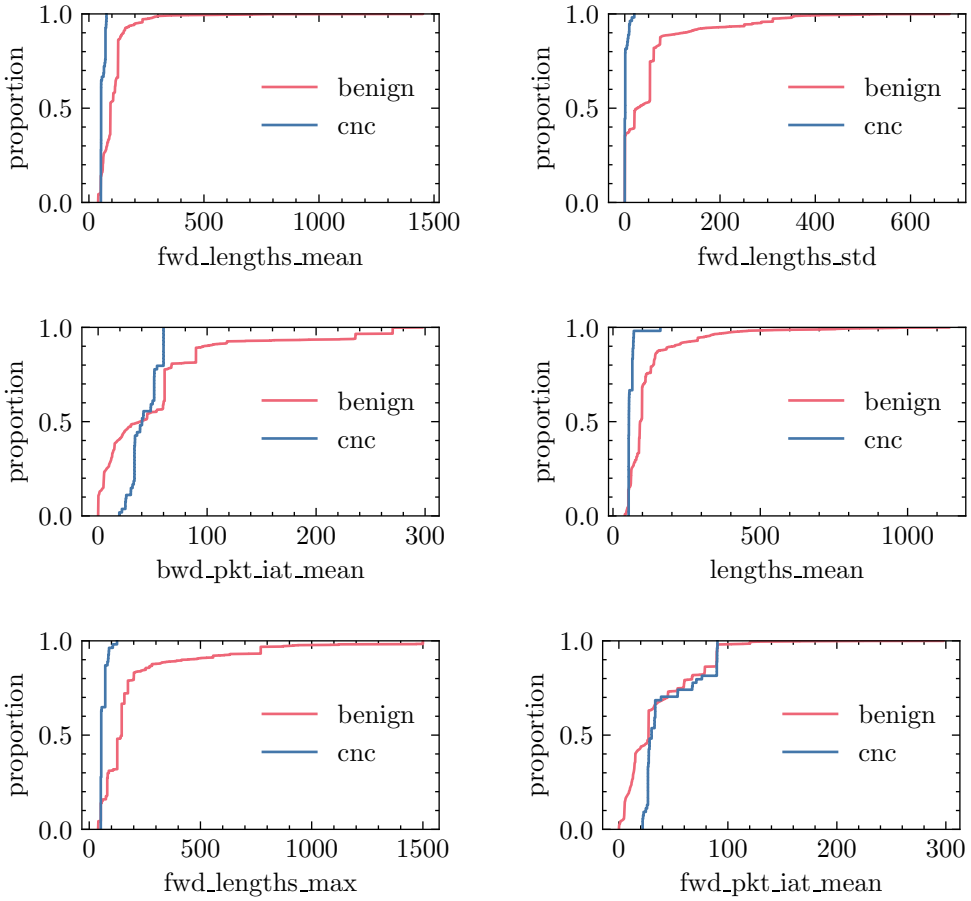

Figure 4.3: Empirical cumulative distribution functions showing how the top six features influencing the logistic regression model can discriminate C&C and benign classes.

metric	**value**
accuracy	0.995 900
precision	0.970 251
recall	0.954 955
F1 score	0.962 543
ROC AUC	0.976 623

Table 4.1: Calculated performance metrics for the baseline model trained on our custom C&C dataset and tested on the IoT-23 dataset.

actual \ predicted	benign	cnc
benign	30370 (0.9983)	52 (0.0017)
cnc	80 (0.045)	1696 (0.955)

Figure 4.4: Confusion matrix of the baseline model. Rows display actual classes of instances, columns display predicted classes. Percentages in cells are normalized over the actual classes (rows).

that works on top of aggregated features extracted from the endpoints' flows within a certain time window.

4.4.1 Time Series

We plan to monitor four distinct univariate time series. All are derived from per-endpoint aggregated features. Aggregation happens only for flows in the outcoming direction; incoming flows are ignored. The four monitored features are: (1) number of unique destination IP addresses, (2) number of unique destination ports, (3) total sent packets, and (4) total sent bytes. We use the number of destination IP addresses to search for horizontal scans. The same goes for vertical scans with the number of destination ports. Total sent packets and total sent bytes are meant to cover DDoS attacks.

We investigated the normal behavior of home devices. Examples of normal behavior for selected devices can be seen in Figure 4.5 and Figure 4.6. In the said figures, we can observe the periodicity of IoT devices' communication compared to user communication on the laptop. Overall, IoT devices generated much less traffic and were contacting only a few external IP addresses. Note that Amazon Echo contacted one to four IP addresses per five minutes, and Belkin Switch was communicating with only one external IP address for four hours straight.

We are convinced that knowing which device is which and monitoring only those devices (which apparently have quite predictable behavior) would make malicious anomaly detection easier. However, we must have in mind that deployment should also be possible in ISP-level networks, where the monitored endpoint represents the whole household. On the same dataset, we simulated this view by aggregating the observed variables of all 19 devices (see Figure 4.7

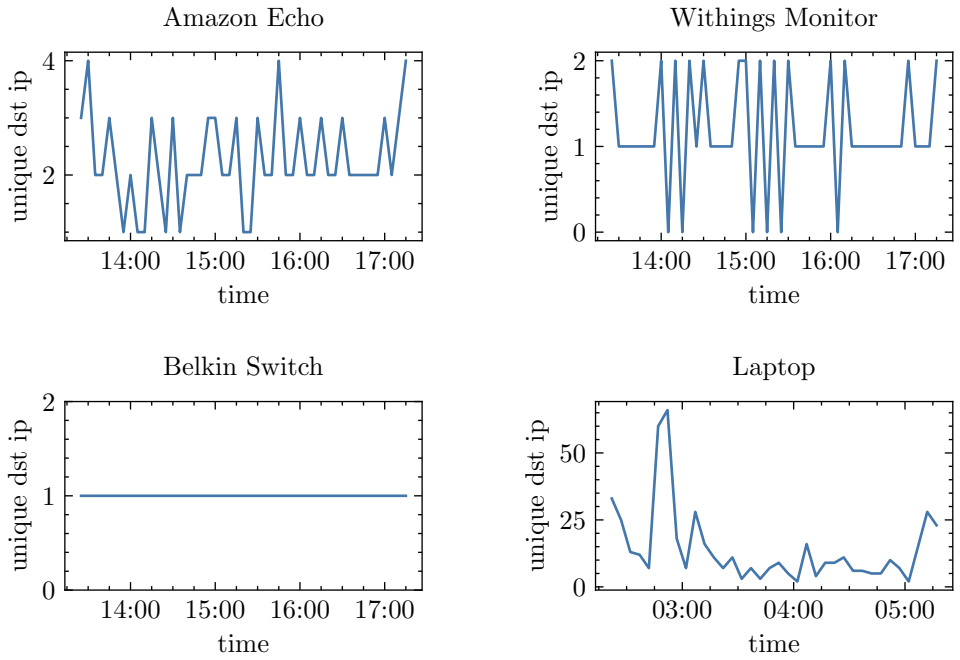

Figure 4.5: Unique destination IP addresses time series of four of the devices present in the UNSW IoT Traces dataset. Unique IP addresses are aggregated within five-minute time windows.

for comparison). If not said otherwise, we will be using this merged view of all devices in our examples.

4.4.2 Forecasting

In principle, time series can be decomposed into a seasonal component, a trend component, and a remainder component. One of the most popular models for time series forecasting, the Autoregressive integrated moving average (ARIMA) model, covers the autocorrelations in the remainder component. The autoregressive (AR) part introduces into the model a linear combination of previous observations. The moving average (MA) similarly influences forecasts using past forecast errors. The integration part describes the degree of the series' differencing. As its main prerequisite, modeling with AR and MA requires the time series to be stationary (time-invariant), so to use the ARIMA model effectively, we need to know the trend levels and seasonality of the series beforehand and differentiate them, respectively [56].

Achieving stationarity is not feasible because not knowing what device is monitored, what is the device's type, or if we can expect any seasonality. This fact would also complicate the maximum likelihood estimation of

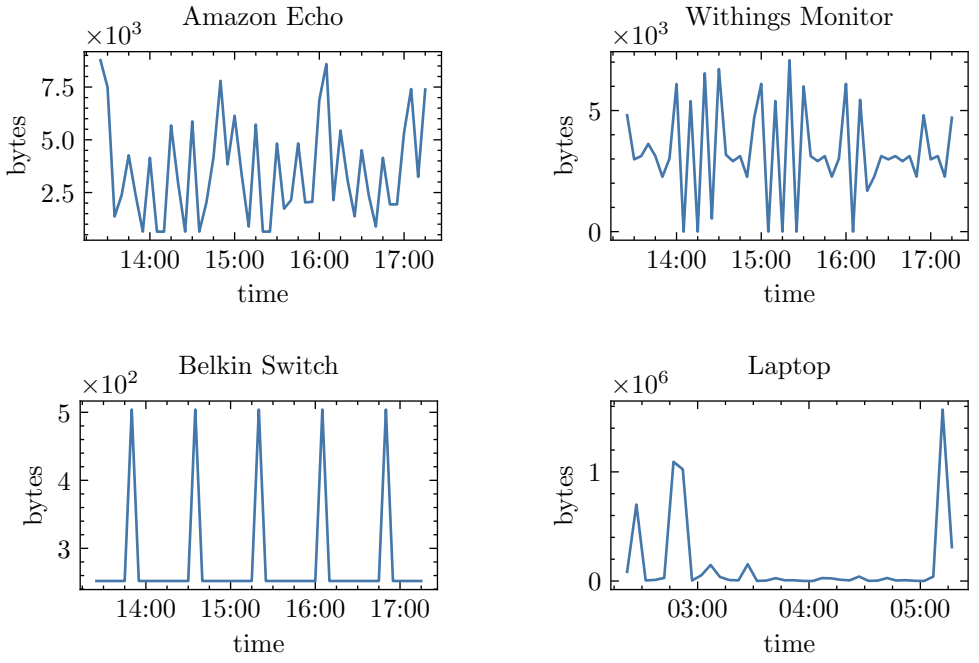

Figure 4.6: Total sent bytes time series of four of the devices present in the UNSW IoT Traces dataset. Total sent bytes are aggregated within five-minute time windows.

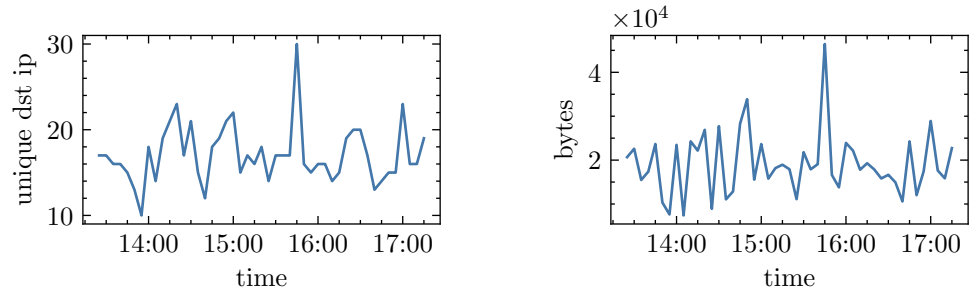

Figure 4.7: Unique destination IP addresses and total sent bytes for the network as a whole. Data was extracted from the first day of UNSW IoT Traces dataset. Presented view aggregates in total of 19 devices which were part of the same network during the network capture.

the ARIMA's parameters, which in our case must be done online, although ARIMA can be represented as a state-space or Bayesian model to address the online estimation. We preferably started with more manageable statistical methods that could be used effectively for online forecasting with low performance and memory footprint. Below, we introduce the considered forecasting models.

Mean and Standard Deviation

If we can assume that the observations have the normal distribution, it may be enough to compare the new observation with thresholds computed as the sample mean (our prediction for the new observation) plus or minus a multiple of the sample standard deviation. This strategy is sometimes called the three-sigma rule [57] referring to:

$$F_X(\mu + 3\sigma) - F_X(\mu - 3\sigma) = \Phi(3) - \Phi(-3) \approx 0.9973 \tag{4.7}$$

The detection module could have specified a different desired multiple of the standard deviation; in our experiments, we used multiples of three and five. We need to ensure the usage of numerically stable algorithms for incremental calculation of sample mean and standard deviation. In [58], there are derived numerically stable recurrent formulae for the mean and variance (this method is also known as Welford's algorithm [59]). The final formula for the mean is:

$$\mu_n = \mu_{n-1} + \frac{1}{n}(x_n - \mu_{n-1}) \tag{4.8}$$

For the incremental standard deviation, we first define $S_n = n\sigma_n^2$. The recurrent formula for S_n (derived again in [58]) is:

$$S_n = S_{n-1} + (x_n - \mu_{n-1})(x_n - \mu_n) \tag{4.9}$$

Finally, we can get the standard deviation as:

$$\sigma_n = \sqrt{\frac{S_n}{n}} \tag{4.10}$$

Quantiles

Similarly, for arbitrary distributions, we can set a threshold to particular estimated quantiles. Methods for quantile estimation are generally based on one of the three principles: (1) sampling the original observations using specialized selection algorithms that try to achieve at most some precision requirement, storing only a low amount of observations, (2) histogram algorithms with fixed bins, providing absolute or relative accuracy, (3) leveraging clustering algorithms, where collections of observations are divided into clusters, keeping only cluster centroids information [60].

In our design, we chose the t-digest [60] method, which belongs to the third category. It keeps an efficient data structure estimating the cumulative distribution function, allowing us to determine the rank-based statistics. During our experiments, we used 99 and 99.9 percentiles as thresholds.

Exponential Smoothing

Brown's simple exponential smoothing [56] offers a trade-off between (1) the next predicted value equal merely to the last observed value and (2) the predicted value equal to the average value. This trade-off is achieved by computing a weighted average influenced by the smoothing parameter $0 \leq \alpha \leq 1$:

$$\hat{y}_{t+1} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \cdots \tag{4.11}$$

written in recurrent form as:

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\hat{y}_t \tag{4.12}$$

Simple exponential smoothing predicts only the level of time series. Exponential smoothing can also be extended to incorporate trend (Holt's exponential smoothing) and seasonality (Holt-Winters' exponential smoothing) [56]. Nevertheless, we made use only of the level predictions as a generic outline for unknown time series. We dynamically compute the threshold based on exponential smoothing prediction intervals as described in [61]. Denoting one-step forecast error as:

$$e_t = y_t - \hat{y}_t \tag{4.13}$$

and estimating the standard deviation of forecast errors using the aforementioned Welford's algorithm. Assuming the one-step forecast errors (e) have the normal distribution, the prediction interval is given as [61]:

$$\hat{y}_t \pm Z_{\alpha/2}\sigma_e \tag{4.14}$$

Choice of α affects how much weight is assigned to more recent values and how quickly the weight coefficients decrease looking to the past. Figure 4.8 shows the thresholding for different values of alpha. We decided to use $\alpha = 0.1$ to enlarge the past values' influence and achieve more smoothed predictions.

4.4.3 Detection Mechanism

Figure 4.9 demonstrates outcomes of the three proposed thresholding methods on the total bytes sent aggregated within five-minute windows for the first eight hours. The figure also shows that some of the spikes present in the normal traffic exceed the thresholds. We considered two approaches to prevent our thresholding methods from reporting regular traffic as malicious.

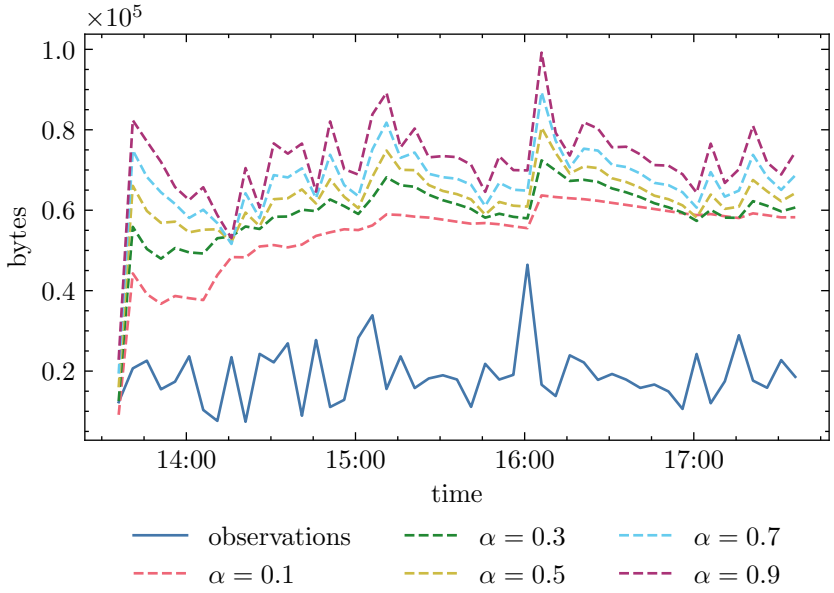

Figure 4.8: Smoothing effects of different values of parameter α for Brown's simple exponential smoothing. All dashed series are generated from the smoothed value plus five prediction error standard deviations.

As a minimal precaution, we combined the dynamic thresholds with a fixed minimum threshold for each aggregated variable, stating the absolute minimal values to be considered for an anomaly.

We also changed the granularity of our aggregations. Instead of aggregating per five-minute time window, we aggregate data five times per one-minute time window, stating that we want to mark the traffic as potentially malicious only if we register anomalies in more consecutive one-minute aggregations. In Figure 4.10, these one-minute aggregation windows are used, and there are no consecutive observations exceeding the thresholds.

4.5 Signature-based Detection

We recognize that some of the behavioral patterns we want to detect are strictly deterministic. Moreover, they are often covered by existing projects, signatures, or protocol parsers. This section briefly shows how we plan to reuse existing network monitoring and threat intelligence knowledge to generate other behavioral indicators from extended IP flows. All the following classifiers could be eventually replaced by more complex deterministic or stochastic methods representing the same indicators.

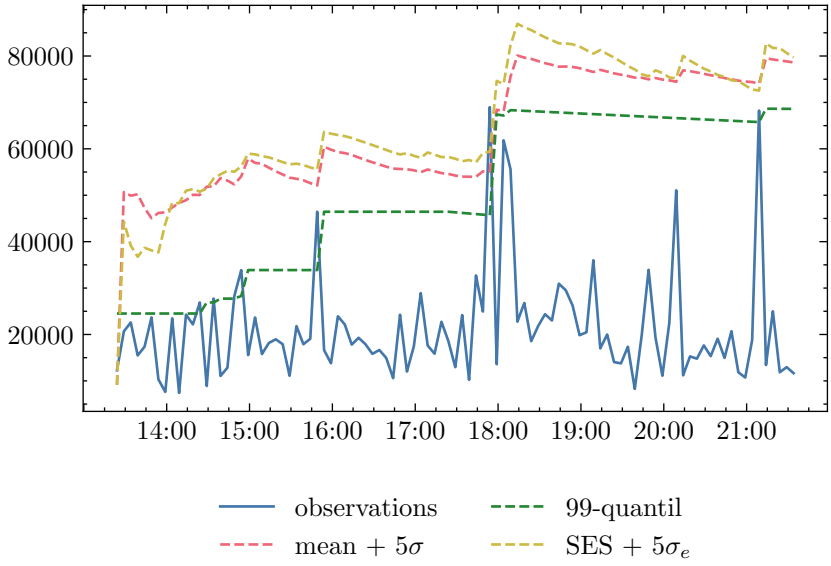

Figure 4.9: Total sent bytes aggregated in five-minute time windows. Displaying three different thresholding methods: (1) mean plus five standard deviations, (2) 99 percentile, (3) Brown's simple exponential smoothing plus five prediction error standard deviations.

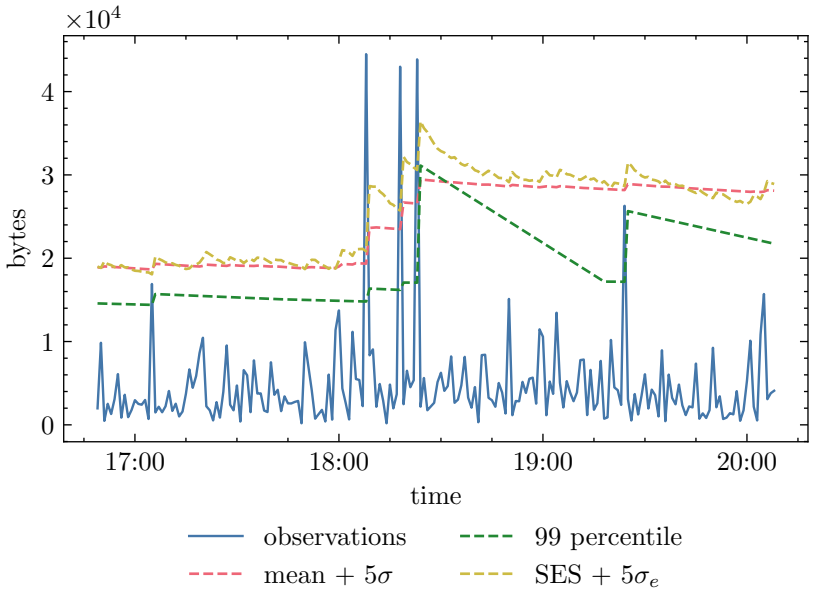

Figure 4.10: Thresholding methods computed using one-minute time windows aggregation. For clarity, the figure displays a close-up view with algorithms already being in place for three hours.

4.5.1 DHT

Some peer-to-peer botnets such as Hajime, or recently very prevalent Mozi, use DHT protocol. Registering the presence of DHT communication can help us detect these botnets, even though DHT by itself does not indicate a malicious event. Based on the level of IP flows' extension, we can realize one or more of the following techniques: (1) detection of port 6881 on UDP transport protocol, (2) detection of bootstrap node IP addresses, (3) detection of bootstrap node domain names inside a DNS request, (4) payload inspection. Port matching does not require us to extend IP flows and can be used to prefilter the traffic. Besides, bootstrap nodes can be found published in public threat intelligence feeds or acquired during malware analysis. For example, Mozi has eight hard-coded bootstrap nodes [43]:

```
dht.transmissionbt.com:6881
router.bittorrent.com:6881
router.utorrent.com:6881
bttracker.debian.org:6881
212.129.33.59:6881
82.221.103.244:6881
130.239.18.159:6881
87.98.162.88:6881
```

We decided to continue with the last detection technique – repurposing existing payload signatures. ET OPEN ruleset has multiple signatures covering different types of messages present in the BitTorrent DHT protocol. The format of the messages is described in [62]. Notice that all bencoded[21] queries start with an ASCII string `d1:ad2:id20:`, and this string is also present in the available signatures. Thanks to this pattern being present at the beginning of the first packet of the flow, it is easy to incorporate this detection mechanism into an extended IP flow-based system sending only the first few bytes of the first packets' payload.

4.5.2 Monero Mining

Even though there were already presented machine learning methods to detect cryptocurrency mining like the one in [63], for the sake of completeness, we will outline some easy-to-implement detection principles the same way as for DHT. As the first option, we can monitor domain requests, looking for the presence of well-known Monero mining pools[22]. Nevertheless, malware authors can register domains resolving to the IP addresses of mining pools to mitigate such detections. More complete public list of cryptomining domains can be

[21] Bencoding is an encoding format used by BitTorrent [62].

[22] `http://moneropools.com/`

found as part of the CoinBlockerLists[23] project. At the time of writing, it lists 162 250 domains.

If we extend IP flows with the first few bytes, we can instead try to detect the underlying Stratum protocol employed for communication between the miner and the mining pool. ET OPEN signatures implement Stratum detection by checking the exact subsequences present in the messages like:

```
|22|id|22 3A|
|22|jsonrpc|22 3A|
|22|method|22 3A 22|login|22|
```

4.5.3 Tor

Addressing malware variants with C&C servers hidden behind the Tor infrastructure, we verify the communication endpoints if any of them is registered as Tor relay node. Various third-party services provide firewall-friendly lists of Tor relays. However, some of them filter only Tor exit relays since some admins want to block the incoming traffic to their services. On the other hand, we are interested in entry relays, to which Tor clients connect as the first hop. Fortunately, information about all the relays can be retrieved via Onionoo[24] REST API served by The Tor Project. Using this API, we keep an updated list of active relays, in which we can search for the flow's destination IP address.

4.6 Combining Classifiers

Finally, as we have designed individual classifiers giving us indicators of different malicious network traffic aspects, we would like to combine them to make decisions as a whole. Motivation lies in the chances of lowering the probability of false positives – assuming the two classifiers are not correlated, we can express their joined probability as a product of the original probabilities. In reality, this assumption presumably does not hold, and the joined probabilities will be scattered somewhere between the original probabilities and their product, still with a fair chance of improving the accuracy.

4.6.1 Informed Meta-Classifiers

Section 4.3.2 already described some approaches of combining classifiers into ensembles. Compared to the random forest or AdaBoost, we are in a situation when we deal with more complicated underlying models, all working on a different set of features. One possible method to deal with separately targeted classifiers is stacking, where the outputs of underlying models are

[23] https://gitlab.com/ZeroDot1/CoinBlockerLists/
[24] https://metrics.torproject.org/onionoo.html

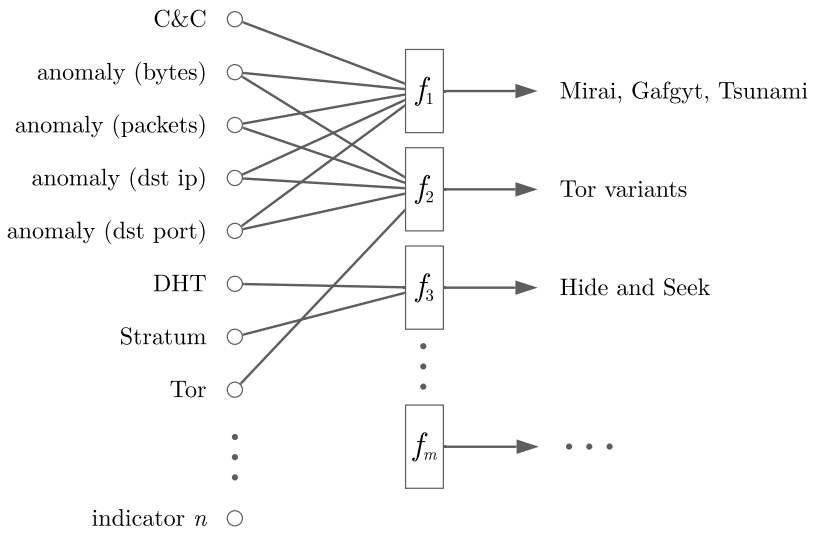

Figure 4.11: Proposed way to combine individual indicators to trigger final detections. Extendible system of n indicators processed by m meta-classifiers.

formed into a feature vector and consumed by a top-level classifier such as logistic regression [52]. Unfortunately, the data our classifiers are processing are so diverse that we can not simply prepare a training dataset to train the top-level classifier.

Instead, since we already reviewed malwares' behavioral characteristics and the most prevalent malware families, we will combine classifiers in an informed manner. Figure 4.11 illustrates the proposed design of combining acquired indicators. In the design, indicators are processed by a set of custom meta-classifiers. We propose two types of informed meta-classifiers – weighted majority voting and Boolean expressions. Overall, this design is extensible so that other flow-based classifiers could produce more indicators in the future. Note that thanks to using Boolean expressions as a meta-classifier, classifiers could also induce inherently benign indicators. In this case, the benign classifier would inform about a particular type of traffic that previously caused false positives, and its output would be negated in the corresponding meta-classifier.

4.6.2 Aggregation

Throughout this chapter, we described classifiers belonging to one of the two general types (see their comparison in Figure 4.12). The anomaly detection classifier aggregates features from the incoming flows by itself, and in every time bin, it produces one indicator. Thus, its output can be processed as is. On the other hand, C&C classifier and signature-based classifiers consume

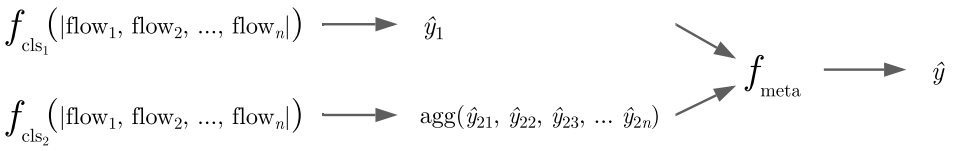

Figure 4.12: Combining two types of classifiers outputs: (1) one result per time bin, (2) n results per time bin (one for each flow).

flows separately and only produce results for those flows. To unify the results, we trigger all detections per time bin. Each time bin consists of several flows, so the results are coming from per-flow classifiers must be aggregated. The aggregate function should be selected based on the semantics of the respective classifier. Typical aggregate functions such as minimum, maximum, average, or count can be used. For the classifiers we designed, we selected the maximum function. Therefore, the C&C classifier will return a positive result if any of the endpoint's flows is detected as a C&C flow.

Chapter 5

Implementation

This chapter briefly describes the implementation portion of the thesis. The central part of the implementation is a software prototype of a detection system called BOTA (BOTnet Analyzer). BOTA realizes the principles designed in the previous chapter inside a standalone system capable of processing extended IP flows produced in the NEMEA infrastructure. BOTA can be deployed inside the NEMEA system or as a standalone tool to analyze both live traffic or pcap files. Finally, we describe the implementation details and the interface of a domain-specific library for exploratory data analysis called FET (Feature Exploration Toolkit), prepared to assist us with data analysis and feature engineering.

5.1 NEMEA Interface and Modules

NEMEA is an open-sourced framework, and its deployment takes the form of multiple interconnected modules. Its modular design allows researchers to prototype new modules without manipulating packet processing, reporting, or system orchestration. Therefore, as developers, we can focus solely on the module's purpose (such as domain-specific filtering, detection, or data analysis), even if we want to implement the module as a separate program [64].

Communication between modules is happening via the framework's TRAP (Traffic Analysis Platform) library. TRAP administers unidirectional interfaces. Each NEMEA module can use zero to multiple input and output interfaces. Processing starts either from a collector module (ipfixcol2) or directly from an exporter (ipfixprobe) – depending on the deployment scale. Both solutions have available outputs in UniRec format. NEMEA core is written in the C programming language, and NEMEA modules can be implemented in C, C++, or Python (as these are the languages with TRAP and UniRec bindings) [64].

Detection modules are a subset of all available NEMEA modules. Their findings are reported in IDEA (Intrusion Detection Extensible Alert) for-

```
{
  "Format": "IDEA0",
  "ID": "936d03e6-662b-4ff0-bdc8-10238d2bcf45",
  "DetectTime": "2021-01-01T14:05:00Z",
  "WinStartTime": "2021-01-01T14:00:00Z",
  "WinEndTime": "2021-01-01T14:05:00Z",
  "Category": ["Intrusion.Botnet"],
  "Description": "IoT Botnet",
  "Source": [
      {
         "Type": ["Botnet"],
         "IP4": ["62.240.165.127"],
      }
  ]
}
```


Figure 5.1: IoT botnet alert in IDEA format. Among the mandatory fields, we report the start and the end of the aggregation window, and the infected device identification – IPv4, IPv6, or MAC address.

mat [65]. As underlying serialization protocol, IDEA uses JSON. Alerts contain four mandatory fields – format version, ID, detection time, and category. IDEA defines 12 categories and 41 subcategories. In our system's implementation, the findings are reported as `Intrusion.Botnet` subcategory. Figure 5.1 shows an example of a generated IDEA alert.

5.2 BOTnet Analyzer

One of the main goals of this thesis was to create a software prototype. Therefore, out of the three options of implementation languages supported by NEMEA (C, C++, and Python), we chose Python. Python's versatility allowed us to quickly prototype the first solution and experiment with pre-trained machine learning models and data analysis modules we developed earlier. In the following sections, we will describe BOTA's modules and classes, its deployment, and testing. Additional details of BOTA are provided in the documentation that can be found either online[25] or on the attached medium.

5.2.1 Implemented Modules

The system consists of six modules: collector, monitor, filter, endpoint, classifier, and anomaly. Representative classes are shown in Figure 5.2.

[25] `https://docs.danieluhricek.cz/bota/`

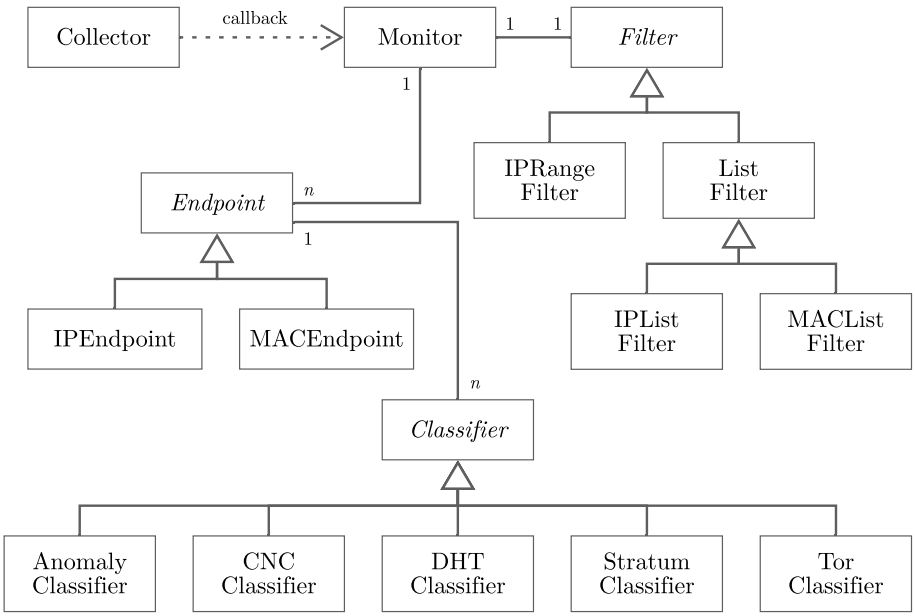

Figure 5.2: Simplified class diagram, which omits few minor classes. Monitor receives messages from collector, applies configured filter, and distributes messages to corresponding endpoints. Endpoint forwards messages to the classifiers based on their type, and applies meta-classification verdict rules.

Collector

Collector class is initialized with a set of TRAP interfaces and a callback function. It parses UniRec messages with the help of pytrap[26] library. Collector creates a thread for each of the TRAP interfaces, transforming received flow records into an internal message format. Messages consist of message types named after the plugin producing the flow record and its data. Additionally, EOF messages are created to signal the end of the metering process. Collector maps the flow record's data is done so that the NEMEA logger (module for storing flow records in CSV file) output is mimicked. Generated messages are sent to the callback function in a critical section.

Monitor

Monitor is the main entry point of the BOTA system. Monitor receives messages and filters them either by IP address or MAC address. It registers the endpoint if it is the first time it notices the endpoint's traffic. It keeps track of time aggregation windows with a fixed predefined interval. After the interval has expired, it collects classification verdicts from the endpoints and reports

[26] `https://pypi.org/project/nemea-pytrap/`

the results in two formats: (1) IDEA alert format, described in the previous section, (2) classification detail format. The latter consists of all positive classification explanations – explanations differ based on the classifier triggering the decision. Moreover, Monitor synchronizes time between classifiers, as time-related properties of the traffic may be critical for some of them (namely, for the time-series anomaly classifier). Note that Monitor keeps track of the time by inspecting timestamps in the processed flow records and does not depend on the present time of operating.

Filter

All filtering classes are accessed via their `apply` method. Four types of filters are implemented: (1) generic lists allowing the selection of any sortable fields for prefiltering, (2) MAC address lists to prefilter endpoints on the home network level where we have data link layer visibility, (3) IP address lists to select smaller subsets of monitored endpoints, and (4) IP ranges to prefilter more extensive networks in CIDR format.

Endpoint

IPEndpoint and MACEndpoint are two specializations of the Endpoint class and are intended to be used with corresponding filtering classes. Endpoint distributes messages to classifiers according to message types. Tor and Anomaly classifiers receive the messages generated by the basic plugin, DHT and Stratum classifiers receive messages generated by the idpcontent plugin, and C&C classifier receives pstats messages. Endpoint gathers classification results from the underlying per-endpoint classifiers and constructs a verdict deciding the endpoint's maliciousness. Current implementation specifies three rules for positive verdicts: (1) positive result from the C&C classifier and at least one positive result from any of the Anomaly classifiers, (2) positive result from the Tor classifier, and again, at least one positive result from the Anomaly classifiers, (3) positive results from both DHT and Stratum classifiers.

Classifier

Classifiers have a unified interface for passing messages and two common attributes: `positive` and `reason`, which represent the positiveness of the decision and classification explanation, respectively. All five classifier classes are implemented in this module:

- DHTClassifier employs prefix match on the hexadecimal encoded data inside idpcontent messages. The reported classification reason consists of flow key and timestamps.

- StratumClassifier matches a set of predefined patterns using the Aho-Corasick algorithm from pyahorasick[27] library. Next, the classifier checks a set of Boolean AND rules to determine which patterns must be matched together to trigger a positive decision. Apart from flow-related information, the decision reason also contains the name of the matched rule.
- TorClassifier examines both source and destination IP addresses. It uses an IPListFilter initialized with a preloaded list of Tor relays.
- CNCClassifier filters out flows with less than three packets or flows shorter than 30 seconds. It then extracts features using the Feature Exploration Toolkit (see Section 5.3) and makes a prediction with a pre-trained model.
- AnomalyClassifier is a generic implementation for anomaly monitoring. It is initialized with a message field to be monitored (e.g., `bytes` or `dst_ip`) and an aggregate function (sum or unique). During prediction, it also takes into account minimum thresholds which are stated in the configuration file. Time processing is similar to the one present in the Monitor class – current time is derived from the seen timestamps. The data is aggregated in one-minute time windows.

Anomaly

Anomaly module contains classes related to time series modeling and anomaly detection. This module is used by AnomalyClassifier mentioned in the previously described model. SimpleExpSmoothing is a class implementing Brown's simple exponential smoothing algorithm and could be eventually replaced with any time series model with the same interface. SimpleExpSmoothing class is using Welford class for online computation of prediction error variances.

5.2.2 Deployment

The main communication interface for BOTA is the same as for other NEMEA modules – TRAP interfaces. Through them, BOTA can be interconnected with any existing part of NEMEA. It can leverage the architecture of ipfixcol2 as a flow collector gathering IPFIX data coming from multiple flow probes, analyze a single network interface using ipfixprobe, or process offline flow records.

Two scripts were prepared for its demonstration; the first one serves to monitor live traffic, the second to analyze pcap files. In both cases, scripts use ipfixprobe to generate flow records, further aggregating pstats records with NEMEA biflow aggregator. The scripts can also be easily accessed inside a provided Docker environment with preinstalled NEMEA and BOTA.

[27] `https://pypi.org/project/pyahocorasick/`

The system accepts a configuration file in JSON format, defining configuration sections for NEMEA, filters, TRAP interfaces, outputs, and models. For the convenience of analyzing short network captures, a user can specify prior knowledge about endpoints'. The prior knowledge is then simulated by generating 300 observations from the Poisson distribution.

5.2.3 Testing

All modules were thoroughly tested. The project contains unit tests for every class, covering 100 % of code statements. Tests for the Collector class depend on the NEMEA traffic repeater module to send flow records from testing TRAP files. Tests were implemented utilizing pytest[28] library. Code coverage was generated by Coverage[29] tool. Refer to the provided documentation for instructions on how to run the tests.

5.3 Feature Exploration Toolkit

During the data analysis process of flows produced by various ipfixprobe plugins, we identified many repetitive tasks. Consequently, we decided to create an extendable framework that would be helpful for future NEMEA projects. The resulting framework, called FET (Feature Exploration Toolkit), aims to minimize the reimplementation of common NEMEA preparation tasks, such as data visualization, preprocessing, aggregation, or feature engineering.

5.3.1 Explorer

The central component of FET is the Explorer class. Explorer provides an API similar to other data analysis and machine learning libraries, preprocessing input data seamlessly. It helps to examine relationships between the target variables and designed features in a typical exploratory data analysis fashion. All visualizations are reached utilizing the Seaborn[30] library. Apart from visualizations and graphs, Explorer evaluates correlated features, estimates feature importances by quickly fitting tree-based models, or grants access to feature scores by computing, e.g., χ^2 statistics.

5.3.2 Feature Modules

Explorer imports modules tied to particular ipfixprobe plugins. For the C&C classifier, we implemented the pstats module. Pstats module and other modules for future ipfixprobe plugins have a common interface consisting of:

[28] `https://docs.pytest.org`
[29] `https://coverage.readthedocs.io`
[30] `https://seaborn.pydata.org/`

- Feature extraction function (`extract_features`), that processes pandas[31] DataFrame and adds feature columns either in place or by returning a new DataFrame.
- Exported list of all feature names (`feature_cols`). The pstats module accounts for 48 features; all of them were already described in Section 4.3.1.
- Directional processing function (`swap_directions`), that can either simulate or handle directional mismatches caused during flow metering.
- Time aggregation function (`aggregate`), that can aggregate shorter flows, induced by early exporting of the flows into several minutes long ones.

FET was adopted by other researchers from Network Monitoring Lab (NETMON[32]) at CTU, working on diverse machine learning problems. Moreover, some of the adopters already implemented their own feature modules – the bstats module for processing flow bursts statistics and the phists module processing histograms of packet sizes and inter-arrival times. For more details and examples, refer to FET's documentation – online[33] or on the attached medium.

[31] `https://pandas.pydata.org/`
[32] `https://netmon.fit.cvut.cz/`
[33] `https://docs.danieluhricek.cz/nemea-fet`

Chapter 6

Evaluation

This chapter presents the results obtained during the evaluation phase. First, we define the evaluation metrics used throughout the sections. Then, we revisit the topic of the C&C classifier and show the performance results of multiple models trained and tuned on an enhanced dataset. The last two sections describe the evaluation of the BOTA system as a whole. For the benign part, we evaluate real-time in CESNET's network – a national e-infrastructure operator and an ISP of many Czech academic institutions. Finally, for the malicious part, we show the detection results of up-to-date malware samples supplied by Avast Software.

6.1 Performance Metrics

In section 4.3.3, we already worked with a few binary classification metrics to support our design decisions. The same metrics will be used throughout this chapter. The direct interpretation classification results can be shown in a so-called confusion matrix. The binary classification confusion matrix is a 2×2 matrix reflecting the number of instances per actual class and per predicted class. Displayed values are denoted as true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN) [66].

Prediction accuracy (ACC) is the first assessed metric. Accuracy and its complementary metric error (ERR = 1 − ACC) define how many percent of samples were correctly classified or misclassified. Accuracy can be expressed in terms of TP, TN, FP, FN, as [66]:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} \tag{6.1}$$

Accuracy can be misleading in the case of highly imbalanced datasets, where positive instances are rare. Hence, we used accuracy combined with other

metrics – precision (PRE), recall (REC), and F1-score (coupling both precision and recall) [66]:

$$PRE = \frac{TP}{TP + FP} \tag{6.2}$$

$$REC = \frac{TP}{FN + TP} \tag{6.3}$$

$$F_1 = 2 \times \frac{PRE \times REC}{PRE + REC} \tag{6.4}$$

Furthermore, solely for the C&C classifier model comparison and hyperparameter tuning, we used the Receiver Operator Characteristics (ROC) graphs which visualize the model's true positive rates with varying decision thresholds and the derived metric – ROC Area Under the Curve (AUC) [66].

6.2 C&C Classifier

During the previously described design and experiment process, we used a training dataset consisting of our custom prepared C&C instances and three days of the UNSW IoT Traces dataset and a testing dataset consisting of IoT-23 C&C instances, IoT-23 benign instances, and 12 days of the UNSW IoT traces dataset. To prepare and evaluate the final C&C classification model, we further extended the dataset with anonymized benign data captured in CESNET's network. The final dataset has a total of 149 270 instances – 147 374 benign instances and 1896 C&C instances. C&C instances account for a little over one percent of the dataset. Table 6.1 reflects the diversity of the dataset by capturing the distribution of destination ports.

60 % of the instances were used for training and 40 % for testing. Similarly to the preparation of the baseline model, we incorporated a pipeline involving oversampling with SMOTE, feature standardization, and a classifier. We tuned the hyperparameters in a five-fold cross-validation procedure, exhaustively searching and generating hyperparameter combinations. The hyperparameters with the best average ROC AUC were selected. The model was retrained on the entire training dataset and evaluated on 59 708 testing instances. Table 6.2 shows the calculated performance metrics for each of the models. The best performing model was AdaBoost (its confusion matrix can be seen in Figure 6.1). The random forest also achieved comparable performance. Decision tree and logistic regression both had significantly lower precision caused by a large number of false positives.

port	count
443	50 096
80	28 243
5228	18 730
8080	7695
5222	6333
993	3939
50443	3838
22	3030
56700	2754
1935	1656

Table 6.1: The top 10 most common destination ports in the final dataset used for the C&C classifier evaluation. Displayed ports cover 84.6 % of the flows in the dataset. The remaining 15.4 % are distributed among more than 10 000 destination ports.

classifier	accuracy	precision	recall	F1-score	ROC AUC
DT	0.989 17	0.531 85	0.982 22	0.690 05	0.985 74
LR	0.993 37	0.652 17	0.984 95	0.784 74	0.989 21
KNN	0.999 63	0.979 70	0.990 42	0.985 03	0.995 08
RF	0.999 80	0.998 61	0.984 95	0.991 73	0.992 47
AB	0.999 85	0.998 62	0.989 06	0.993 81	0.994 52

Table 6.2: Calculated performance metric for the following classifiers: decision tree (DT), logistic regression (LR), k-nearest neighbors (KNN), random forest (RF), and AdaBoost (AB).

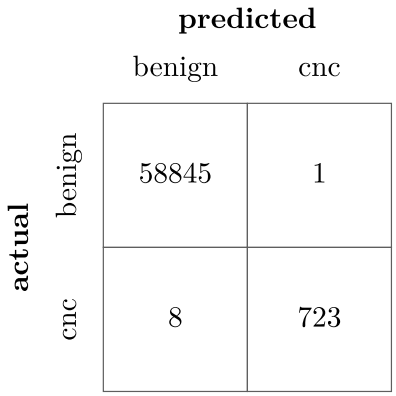

Figure 6.1: Confusion matrix of the final AdaBoost classifier.

property	value
flows	9 379 169
packets	481 148 442
packet sizes	229.86 GB
duration	39.163 hours

Table 6.3: Volumetric summaries of the CESNET capture.

positive result	count
DHT	543
Tor	303
anomaly (packets)	31
anomaly (bytes)	14
anomaly (dst port)	6
anomaly (dst ip)	2

Table 6.4: Positive results for individual classifiers evaluated on the CESNET capture.

6.3 CESNET Traffic

To verify if the proposed detection mechanism is viable in real scenarios producing a minimum amount of false-positive alerts, we tested the BOTA system on long flow captures in CESNET's network. We selected a `/24` subnet of public IP addresses. Each such IP address can represent multiple endpoints masqueraded behind NAT. Further properties of the capture are displayed in Table 6.3. BOTA was able to process all flows in about four hours.

We noticed a total of 899 separate positive classifiers' results, caused mainly by µTorrent DHT users and by the presence of Tor IP relays; we additionally noticed few positive anomaly results. Specific numbers of positive results are listed in Table 6.4. We also examined 17 cases when more than one classifier returned positive results (see Table 6.5 for their enumeration). However, no such combination of results was consistent with meta-classification rules; therefore, they did not cause any alerts. The results we analyzed were correct, and the detection algorithm behaved as expected.

6.4 Avast Malware Captures

Finally, we proceeded with the evaluation on actual malware samples. To demonstrate the usability of the designed concept on current up-to-date malicious traffic, we limited ourselves only to real-world malware samples that

combination	count
anomaly (bytes) + anomaly (packets)	13
anomaly (dst ip) + anomaly (dst port)	2
DHT + Tor	2

Table 6.5: Combinations of positive results on the CESNET capture. As might be expected, anomaly results for sent bytes are correlated with the results for sent packets. The same goes for the results of the unique destination IP addresses and the unique destination ports.

were active in April 2021. The overall malware evaluation process can be described as follows:

1. We downloaded an example list of 500 malware files, and their corresponding short-term pcap captures provided by Avast Software. Selected files had at least one hit by any static or behavioral YARA rule indicating its potential maliciousness.

2. We created a subset of 105 pcaps sufficient for our needs. As might be expected, not all pcaps contained necessary information – in various cases, the C&C server was not active at the time of capture since C&C servers accessed directly via their IP address can have a lifespan of only a few days.

3. We manually annotated the network behavior present in the pcaps. The initial analysis of pcaps was done using LiSa sandbox, which also has the functionality to analyze pcaps, retrieve information about resolved domains (potential C&C servers, mining pools, or DHT nodes), location of IP addresses, or scanning anomalies.

4. Annotated pcaps were processed using BOTA's pcap monitoring script. We compared the detection results with annotations and manually validated all detection reasons – compared the hosts detected as C&C servers with the actual annotated C&C servers, confirmed anomaly counters, and closely inspected all flows detected by signature-based classifiers.

6.4.1 C&C Classifier Results

For the C&C classifier, we got the results from 83 analyses – these 83 out of 105 analyses held active client-server C&C communication targeted by this classifier. In the stated samples, there were 63 malware variants with binary C&C protocol, 18 variants with custom text-based C&C protocol, and three

variants using IRC. We were able to identify some previously described variants of Mirai and Gafgyt, such as Fbot or Airdropbot.

The classifier correctly detected 77 out of 83 C&C communication flows. All variants using binary C&C protocols, primarily derived from Mirai, were correctly identified. For text-based C&C protocols, 16 were correctly identified, and two were missed. The worst performance was measured for IRC-based variants with only one positive detection. Nevertheless, we found this behavior as expected because the evaluated pcaps contained mostly one evaluation window. In the case of IRC, the first window holds welcome messages and initialization compared to the heartbeat messages and commands we wanted to detect, which would be present in later evaluation windows.

The classifier was able to generalize trained information and detect diverse communication patterns. Out of 77 correctly identified variants, we observed 42 unique reassembled TCP streams. The most prevalent streams obeyed the base Mirai implementation. Moreover, the classifier recognized 37 unique C&C servers. The servers were in 48 pcaps accessed directly via their IP addresses, and in the remaining 29 pcaps, the malware first resolved a C&C domain.

Our findings indicate that statistical analysis of inter-arrival times, packet lengths, payload correlation, or payload entropy is sensitive to TCP congestion mechanisms, TCP retransmission, and TCP keep-alive packets. Such packets can be filtered out to unify our view of the captured traffic. As a reference, we can follow Wireshark's implementation of TCP sequence and acknowledge numbers analysis [67]. All of these issues are being resolved directly in the pstats plugin of ipfixprobe exporter. Compared to the one used in this thesis, the latest implementation analyzes packet lengths beginning after the transport layer; it already eliminates zero-length packets and TCP retransmissions. We encountered only one false positive in the whole evaluation dataset, and it was caused by the handling of repeated flows that used the same port numbers. The evaluated version of ipfixprobe aggregated packets strictly by the flow key, ignoring flow termination by TCP FIN. This particular issue was also recently fixed, and ipfixprobe can distinguish such separate connections.

6.4.2 Anomaly Results

Scanning or DDoS was present in 95 of the analyzed pcaps. As was previously mentioned, BOTA accepts the configuration of prior knowledge by stating λ parameters for observations' Poisson distributions. The prior configurations, together with minimum anomaly thresholds, are listed in Table 6.6.

The algorithm correctly identified all anomalies. Besides, in all of the 95 cases, anomalies were reported for multiple observed values. Specific combinations of triggered anomalies can be seen in Table 6.7. Furthermore, we examined how many prediction and threshold anomalies were registered in consecutive one-minute windows. We configured a 500 seconds monitoring

property	value
λ (dst ip)	5
λ (dst port)	5
λ (packets)	30
λ (bytes)	1000
min threshold (dst ip)	20
min threshold (dst port)	20
min threshold (packets)	500
min threshold (bytes)	10 000

Table 6.6: Configuration of anomaly classifier used for the Avast malware dataset evaluation. Endpoint represents only one host, with non-frequent communication.

anomaly combination	count
bytes + packets + dst ip	82
bytes + packets + dst ip + dst port	11
bytes + packets	2

Table 6.7: Combinations of positive results from different anomaly classifiers. The most common combination expresses scanning routines. Anomalous amounts of destination ports were reported thanks to many DHT peers. Two pcaps triggering bytes and packets anomalies held only DDoS traffic.

observed	min	max	median
dst ip	1008	314 924	20 860
dst port	45	119	75
packets	1481	421 768	27 397
bytes	117 420	21 807 800	1 425 700

Table 6.8: Minimum, maximum, and median of reported anomalous values. Reported anomalous values are always the maxima of one-minute aggregations among the overall decision window.

interval in our setup; therefore, anomaly classifiers could process up to eight one-minute windows. In about 90 % of all reported anomalies, at least five consecutive time windows exceeded both the prediction threshold (set by the exponential smoothing prediction interval) and the minimum anomaly threshold.

6.4.3 Signature-based Classifiers Results

The evaluation dataset carried 10 Monero miners from unspecified IoT families with a low number of antivirus detections at the time of evaluation. Their captured pcaps contained unique TCP streams utilizing the Stratum protocol, and the Stratum login rule correctly identified all of them. DHT traffic was generated during the analyses of 11 samples, all belonging to the Mozi family. All of them produced unique TCP streams, and again, all were correctly identified by the DHT classifier. None of the evaluated malware samples did use Tor communication. However, some Tor relay nodes were randomly contacted as part of the scanning; therefore, the Tor classifier returned a positive result. This behavior may or may not be desired, and according to our preferences, we could declare minimum thresholds for the number of packets, bytes, or flow duration to narrow possible Tor detections.

6.4.4 Meta-Classifiers Discussion

In this thesis, we primarily targeted IoT malware families with client-server C&C architecture via a meta-classifier incorporating both the C&C classifier and the anomaly classifier. The evaluation confirmed that this meta-classifier could be more narrow, leveraging that all positive anomaly results were accompanied by at least one more. Similarly, we could require more consequent positive results for the one-minute windows. Overall, the combinations of classification results are shown in Table 6.9.

From the resulting combinations, we are able to deduce new meta-classifier functions. Mozi malware family triggered combinations of DHT and all four measured anomalies – the anomalous number of destination ports due to its numerous peers hosted on random ports and the remaining anomalous values caused by its scanning routine. Generic Monero miners could also be detected based on the Stratum protocol and triggered anomalies during the scanning. Such new meta-classifier functions demonstrate the benefits of the proposed detection architecture that can consume more independent indicators.

combination	**count**
cnc + anomaly (bytes + packets + dst ip)	52
cnc + anomaly (bytes + packets + dst ip) + tor	16
cnc + anomaly (bytes + packets)	2
cnc	8
dht + anomaly (bytes + packets + dst ip + dst port)	9
dht + anomaly (bytes + packets + dst ip + dst port) + tor	2
stratum + anomaly (bytes + packets + dst ip)	7
stratum + anomaly (bytes + packets + dst ip) + tor	3
anomaly (bytes + packets + dst ip)	6

Table 6.9: Combination of positive results on the Avast malware dataset.

Conclusion

This thesis focused on the problematics of IoT malware, analyzing its network behavioral patterns and detection possibilities in flow-based monitoring systems. Current available IoT datasets were reviewed and supplemented with custom C&C data produced in a controlled, virtualized environment. Furthermore, we designed, implemented, and tested the detection methods of various aspects of IoT malware communication. A novel approach of combining previously acquired network indicators was presented, illustrating its extensibility to cover the presence of new malware families. The final implementation is able to process both live network traffic and network captures, leveraging the existing parts of NEMEA.

As one of the main contributions of this thesis, we studied machine learning methods applied to extended IP flows to recognize IoT malware C&C communication. We justified the feasibility of the proposed methodology and selected features by fitting a baseline model to the custom C&C dataset, and evaluating on the considerably more diverse IoT-23 dataset. The baseline model achieved the accuracy of 99.5 % and the precision of 97 %. It correctly classified malware families not used during the training phase, successfully generalizing common aspects of the C&C communication. The final model was trained and evaluated on parts of the joined dataset consisting of CESNET anonymized benign flows, UNSW IoT Traces benign flows, IoT-23 C&C flows, and the C&C flows coming from our custom dataset. The best performing model was AdaBoost with the accuracy of 99.9 % and the precision of 99.8 %.

Further, we considered methods for online univariate anomaly detection, selecting Brown's simple exponential smoothing as a mere model for the sum of sent bytes, the number of transmitted packets, unique contacted IP addresses, and unique destination ports. The implemented generic algorithm utilizing the models' predictions correctly recognized anomalies pointing to malicious scanning, DDoS, and, surprisingly, peer-to-peer behavior of the Mozi malware family. In addition, both anomaly and C&C classifiers were accompanied by

signature-based detections, matching on the first n bytes at the beginning of the extended IP flow.

Our results clearly illustrate the potential of statistical data analysis in a flow-based environment but also raise the question of how unpredictable or faulty nuances in network traffic (such as TCP retransmission packets) should be formally represented. Our findings motivated recent changes in the CESNET ipfixprobe exporter, changing its definitions and implementation for packets included in pstats plugin processing.

The extensible architecture of the proposed detection algorithm is open for future research, leaving space for other indicators to be incorporated. Following the same direction presented in this thesis, future studies could investigate, for instance, the consequences of domain generation algorithms or exploitation of HTTP services. Future research could also possibly dig deeper into machine learning algorithms for text analysis applied to the first n bytes of the flow produced by the idpcontent plugin, recognizing the unique properties of IoT malware-specific application-layer payloads.

Bibliography

[1] Kumar, D.; Shen, K.; et al. All Things Considered: An Analysis of IoT Devices on Home Networks. In *28th USENIX Security Symposium (USENIX Security 19)*, USENIX Association, Aug. 2019, ISBN 978-1-939133-06-9, pp. 1169–1185. Available from: https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-deepak

[2] Cozzi, E.; Graziano, M.; et al. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 161–175, doi: 10.1109/SP.2018.00054.

[3] Xu, C.; Chen, S.; et al. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys & Tutorials*, volume 18, no. 4, 2016: pp. 2991–3029, doi:10.1109/COMST.2016.2566669.

[4] Wang, X.; Hong, Y.; et al. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA: USENIX Association, Feb. 2019, ISBN 978-1-931971-49-2, pp. 631–648. Available from: https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[5] Ghafir, I.; Prenosil, V.; et al. A Survey on Network Security Monitoring Systems. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016, pp. 77–82, doi: 10.1109/W-FiCloud.2016.30.

[6] Prithi, S.; Sumathi, S.; et al. A survey on intrusion detection system using deep packet inspection for regular expression matching. *International Journal of Electronics, Electrical and Computational System*, volume 6, no. 1, 2017.

[7] *Suricata User Guide – Suricata 6.0.1 documentation* [online]. 2020, [cit. 2021-02-25]. Available from: https://suricata.readthedocs.io/en/suricata-6.0.1/

[8] *Zeek Documentation: Book of Zeek* [online]. c2019-2021, [cit. 2021-02-25]. Available from: https://docs.zeek.org/

[9] Mockapetris, P. *Domain names – Implementation and specification*. RFC 1035, RFC Editor, November 1987. Available from: http://www.rfc-editor.org/rfc/rfc1035.txt

[10] Uhříček, D. *Multiplatform Linux Sandbox for Analyzing IoT Malware*. Bachelor's thesis, Brno University of Technology, Faculty of Information Technology, 2019. Available from: https://www.fit.vut.cz/study/thesis/22120/

[11] Choi, H.; Lee, H.; et al. Botnet Detection by Monitoring Group Activities in DNS Traffic. In *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, 2007, pp. 715–720, doi: 10.1109/CIT.2007.90.

[12] Fielding, R.; Reschke, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231, RFC Editor, June 2014. Available from: http://www.rfc-editor.org/rfc/rfc7231.txt

[13] Althouse, J. *TLS Fingerprinting with JA3 and JA3S* [online]. 2019, [cit. 2021-03-12]. Available from: https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967

[14] Hoffman, P.; McManus, P. *DNS Queries over HTTPS (DoH)*. RFC 8484, RFC Editor, October 2018. Available from: http://www.rfc-editor.org/rfc/rfc8484.txt

[15] Hamilton, R.; Iyengar, J.; et al. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-hamilton-quic-transport-protocol-00, IETF Secretariat, July 2016. Available from: https://tools.ietf.org/pdf/draft-hamilton-quic-transport-protocol-00.pdf

[16] Hofstede, R.; Čeleda, P.; et al. Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX. *IEEE Communications Surveys & Tutorials*, volume 16, no. 4, 2014: pp. 2037–2064, doi:10.1109/COMST.2014.2321898.

[17] Sperotto, A. *Flow-based intrusion detection*. Dissertation thesis, University of Twente, Netherlands, Oct. 2010, doi:10.3990/1.9789036530897.

[18] *ipfixprobe – IPFIX flow exporter* [online]. [cit. 2021-03-12]. Available from: https://github.com/CESNET/ipfixprobe

[19] *IPFIXcol2* [online]. 2020, [cit. 2021-03-12]. Available from: https://github.com/CESNET/ipfixcol2

[20] Quittek, J.; Bryant, S.; et al. *Information Model for IP Flow Information Export*. RFC 5102, RFC Editor, January 2008. Available from: http://www.rfc-editor.org/rfc/rfc5102.txt

[21] David McGrew, P. P., Blake Anderson; Hudson, B. *Cisco Joy* [online]. 2016, [cit. 2021-03-12]. Available from: https://github.com/cisco/joy

[22] Inacio, C. M.; Trammell, B. YAF: Yet Another Flowmeter. In *Proceedings of the 24th International Conference on Large Installation System Administration*, LISA'10, USENIX Association, 2010, pp. 107–118.

[23] Draper-Gil, G.; Lashkari, A. H.; et al. Characterization of Encrypted and VPN Traffic using Time-related Features. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy*, INSTICC, SciTePress, 2016, ISBN 978-989-758-167-0, ISSN 2184-4356, pp. 407–414, doi:10.5220/0005740704070414.

[24] Habibi Lashkari., A.; Draper-Gil., G.; et al. Characterization of Tor Traffic using Time based Features. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, SciTePress, 2017, ISBN 978-989-758-209-7, ISSN 2184-4356, pp. 253–262, doi:10.5220/0006105602530262.

[25] Meidan, Y.; Bohadana, M.; et al. N-BaIoT: Network-Based Detection of IoT Botnet Attacks Using Deep Autoencoders. *IEEE Pervasive Computing*, volume 17, no. 3, 2018: pp. 12–22, doi:10.1109/MPRV.2018.03367731.

[26] Celik, Z. B.; Raghuram, J.; et al. Salting Public Traces with Attack Traffic to Test Flow Classifiers. In *Proceedings of the 4th Conference on Cyber Security Experimentation and Test*, CSET'11, USENIX Association, 2011.

[27] Marzano, A.; Alexander, D.; et al. The Evolution of Bashlite and Mirai IoT Botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 813–818, doi:10.1109/ISCC.2018.8538636.

[28] Cozzi, E.; Vervier, P.-A.; et al. The Tangled Genealogy of IoT Malware. In *Annual Computer Security Applications Conference*, ACSAC '20, Association for Computing Machinery, 2020, ISBN 9781450388580, doi:10.1145/3427228.3427256.

[29] Hilt, S.; Mercês, F.; et al. *Worm War: The Botnet Battle for IoT Territory* [online]. 2020, [cit. 2021-03-28]. Available from: https://documents.trendmicro.com/assets/white_papers/wp-worm-war-the-botnet-battle-for-iot-territory.pdf

[30] Symantec. *Internet Security Threat Report – April 2017* [online]. 2017, [cit. 2021-03-28]. Available from: https://docs.broadcom.com/doc/istr-22-2017-en

[31] Antonakakis, M.; April, T.; et al. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, ISBN 978-1-931971-40-9, pp. 1093–1110. Available from: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[32] Dulaunoy, A.; Wagener, G.; et al. An extended analysis of an IoT malware from a blackhole network. In *Proceedings of the Networking Conference TNC*, volume 17, 2017.

[33] Margolis, J.; Oh, T. T.; et al. An In-Depth Analysis of the Mirai Botnet. In *2017 International Conference on Software Security and Assurance (ICSSA)*, 2017, pp. 6–12, doi:10.1109/ICSSA.2017.12.

[34] Edwards, S.; Profetis, I. *Hajime: Analysis of a decentralized internet worm for IoT devices* [online]. Technical report, Rapidity Networks, Oct. 2016, [cit. 2021-03-29]. Available from: https://security.rapiditynetworks.com/publications/2016-10-16/hajime.pdf

[35] Herwig, S.; Harvey, K.; et al. Measurement and Analysis of Hajime, a Peer-to-peer IoT Botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, Jan. 2019, doi:10.14722/ndss.2019.23488.

[36] Şendroiu, A.; Diaconescu, V. Hide'n'seek: An adaptive peer-to-peer IoT botnet. In *2018 Virusbulletin Conference*, 2018, [cit. 2021-03-29]. Available from: https://www.virusbulletin.com/uploads/pdf/magazine/2018/VB2018-Sendroiu-Diaconescu.pdf

[37] Středa, A.; Neduchal, J. *Let's play Hide 'N Seek with a botnet* [online]. Dec. 2018, [cit. 2021-03-29]. Available from: https://blog.avast.com/hide-n-seek-botnet-continues

[38] Botezatu, B. *New Hide 'N Seek IoT Botnet using custom-built Peer-to-Peer communication spotted in the wild* [online]. Jan. 2018, [cit. 2021-03-29]. Available from: https://labs.bitdefender.com/2018/01/new-hide-n-seek-iot-botnet-using-custom-built-peer-to-peer-communication-spotted-in-the-wild

[39] Křoustek, J.; Iliushin, V.; et al. *Torii botnet - Not another Mirai variant*, [online]. Sept. 2018, [cit. 2021-03-30]. Available from: https://blog.avast.com/new-torii-botnet-threat-research

[40] *The new developments Of the FBot* [online]. Feb. 2019, [cit. 2021-04-10]. Available from: https://blog.netlab.360.com/the-new-developments-of-the-fbot-en/

[41] *Gafgtyt_tor and Necro are on the move again* [online]. Feb. 2021, [cit. 2021-04-10]. Available from: https://blog.netlab.360.com/gafgtyt_tor-and-necro-are-on-the-move-again/

[42] Zobal, L.; Kolář, D.; et al. Current State of Honeypots and Deception Strategies in Cybersecurity. In *2019 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, 2019, doi:10.1109/ICUMT48472.2019.8970921.

[43] *Mozi, Another Botnet Using DHT* [online]. Dec. 2019, [cit. 2021-04-11]. Available from: https://blog.netlab.360.com/mozi-another-botnet-using-dht/

[44] Douligeris, C.; Mitrokotsa, A. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, volume 44, no. 5, 2004: pp. 643–666, ISSN 1389-1286, doi:10.1016/j.comnet.2003.10.003.

[45] Pastrana, S.; Suarez-Tangil, G. A First Look at the Crypto-Mining Malware Ecosystem: A Decade of Unrestricted Wealth. In *Proceedings of the Internet Measurement Conference*, IMC '19, Association for Computing Machinery, 2019, ISBN 9781450369480, p. 73–86, doi:10.1145/3355369.3355576.

[46] Ring, M.; Wunderlich, S.; et al. A survey of network-based intrusion detection data sets. *Computers Security*, volume 86, 2019: pp. 147–167, ISSN 0167-4048, doi:https://doi.org/10.1016/j.cose.2019.06.005.

[47] Hussain, F.; Hussain, R.; et al. Machine Learning in IoT Security: Current Solutions and Future Challenges. *IEEE Communications Surveys & Tutorials*, volume 22, no. 3, 2020: pp. 1686–1721, doi:10.1109/COMST.2020.2986444.

[48] Sivanathan, A.; Gharakheili, H. H.; et al. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*, volume 18, no. 8, 2019: pp. 1745–1759, doi:10.1109/TMC.2018.2866249.

[49] Parmisano, A.; Garcia, S.; et al. *A labeled dataset with malicious and benign IoT network traffic.* [online]. Jan. 2020, Stratosphere Laboratory

[cit. 2020-10-01]. Available from: https://www.stratosphereips.org/datasets-iot23

[50] Garcia, S.; Grill, M.; et al. An empirical comparison of botnet detection methods. *Computers & Security*, volume 45, 2014: pp. 100–123, ISSN 0167-4048, doi:10.1016/j.cose.2014.05.011.

[51] Molnar, C. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. 2019. Available from: https://christophm.github.io/interpretable-ml-book/

[52] Raschka, S.; Mirjalili, V. *Python Machine Learning: Machine Learning and Deep Learnig with Python, scikit-learn, and TensorFlow*. Packt Publishing, second edition, 2017, ISBN 978-1-78712-593-3.

[53] Shalev-Shwartz, S.; Ben-David, S. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014, ISBN 978-1-107-05713-5.

[54] Laurent, H.; Rivest, R. L. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, volume 5, no. 1, May 1976: pp. 15–17.

[55] Chawla, N. V.; Bowyer, K. W.; et al. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, volume 16, Feb. 2002: pp. 321–357.

[56] Hyndman, R. J.; Athanasopoulos, G. *Forecasting: principles and practice*. OTexts, 2018, [cit. 2021-04-20]. Available from: https://otexts.com/fpp2/

[57] Hochenbaum, J.; Vallis, O. S.; et al. Automatic Anomaly Detection in the Cloud Via Statistical Learning. *arXiv preprint arXiv:1704.07706*, 2017.

[58] Finch, T. *Incremental calculation of weighted mean and variance* [online]. Feb. 2009, [cit. 2021-04-20]. Available from: https://fanf2.user.srcf.net/hermes/doc/antiforgery/stats.pdf

[59] Knuth, D. E. *Art of computer programming, volume 2: Seminumerical algorithms*. Addison-Wesley, 2014, ISBN 9780201038224.

[60] Dunning, T. The t-digest: Efficient estimates of distributions. *Software Impacts*, volume 7, 2021: p. 100049, ISSN 2665-9638, doi:10.1016/j.simpa.2020.100049.

[61] Yar, M.; Chatfield, C. Prediction intervals for the Holt-Winters forecasting procedure. *International Journal of Forecasting*, volume 6, no. 1, 1990: pp. 127–137, ISSN 0169-2070, doi:10.1016/0169-2070(90)90103-I.

[62] Loewenstern, A.; Norberg, A. *DHT Protocol* [online]. Jan. 2008, [cit. 2021-04-21]. Available from: http://www.bittorrent.org/beps/bep_0005.html

[63] Veselý, V.; Žádník, M. How to detect cryptocurrency miners? By traffic forensics! *Digital Investigation*, volume 31, 2019, ISSN 1742-2876, doi: 10.1016/j.diin.2019.08.002.

[64] Cejka, T.; Bartos, V.; et al. NEMEA: A Framework for Network Traffic Analysis. In *2016 12th International Conference on Network and Service Management (CNSM)*, 2016, pp. 195–201, doi:10.1109/CNSM.2016.7818417.

[65] *Intrusion Detection Extensible Alert* [online]. 2017, [cit. 2021-04-25]. Available from: https://idea.cesnet.cz

[66] Raschka, S. An Overview of General Performance Metrics of Binary Classifier Systems. *arXiv preprint arXiv:1410.5330*, 2014.

[67] *Wireshark User's Guide: Version 3.5.0* [online]. 2021, [cit. 2021-04-28]. Available from: https://www.wireshark.org/docs/wsug_html_chunked/

Appendix A

Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BGP Border Gateway Protocol

C&C Command and Control

DDoS Distributed Denial of Service

DHT Distributed Hash Table

DNS Domain Name System

DPI Deep Packet Inspection

DVR Digital Video Recorder

ECDSA Elliptic Curve Digital Signature Algorithm

FPGA Field-Programmable Gate Array

FTP File Transfer Protocol

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IE Information Element

IP Internet Protocol

IPFIX IP Flow Information Export

IRC Internet Relay Chat

ISP Internet Service Provider

JSON JavaScript Object Notation

MPLS Multiprotocol Label Switching

NIDS Network Intrusion Detection System

NTP Network Time Protocol

PCA Principal Component Analysis

RCE Remote Code Execution

SIMD Single Instruction, Multiple Data

SIP Session Initiation Protocol

SMTP Simple Mail Transfer Protocol

SSDP Simple Service Discovery Protocol

SSH Secure Shell Protocol

TCP Transmission Control Protocol

TFTP Trivial File Transfer Protocol

TLS Transport Layer Security

TTL Time to Live

UDP User Datagram Protocol

URL Uniform Resource Locator

VLAN Virtual Local Area Network

VPN Virtual Private Network

XML Extensible Markup Language

Appendix B

Evaluated Malware Samples

sha256
e07b81c2c2dec55e174cb3d50a59cdb0e605f2e64ac345e2b1bf5a20ae2ba38d
1feaa33de1f721891d3f7c9e2b3bb114f6d58205a802697a54be7e4814c4e0c8
2c0e69f96fb5e0eb71e6ad82aa54d1be78230bdbfb4869f0d50ea9624a002b87
3c9c18b43b098cf682d4c5223a582566e3f41293d3eb2c7ce978f66989231794
5444a17585311af11f280c5e064e91e78c270cb65f5c407262231e1e4b63d79c
494028ff917fb035b12104a3e3a930304b258093fc239c7833f99b8d23060b4d
8d5f9a3776a256af2b866799612bb2b3da0b0f07629ebeea16ec5c281eef2daf
31bfb20b0341bf36e7fef79b42b45261c878cbe88c38cef06f6cd58a4695a3e8
f2cd1d0cd5bf18e9b20016c5c578386e3e6fef7b16f74c766d335e78e8fd56c6
d42de972c530191f1a590104f59f5e55ecc7539caaa9693c45b709f4010c69f9
0dc15850d6952be4feb2f94d847a7f4358689ac155fe8cd8a5777772ab2f0312
564b8b112d12aa11225bb996955e0a869094c68b3b76ef1a2391d04d335d91a8
6ddb78a5799609f5ae439a3270e99e1cfb48f6b64a271449c29f9616c00bcd9c
00599ac9c323a6e8f87ecd9fbfe4177cf86e9fb59091644f8053512eea99d6e9
8f3448bc933f62d7f759961cfff976cd815c415fbfdd20c4d330bb33515ce54a
0558a580d1bdcfcf286e3c32b13cb42fdb81ca1fae9fb3f4db03063b28d07e7e
e6ebf6d2a4f171f54c49cdffd4ecdbb11b33a8d7e0a1e50fd63c97366194ee51
4dd3619f08167cabe2633db1cde531c44e3a997f4899e99f7b1d7d117e620394
4e19d1b15c549b8af5e26f0935fc7ced07c6b1eda47909e54f5d2f258b673431
664fd512e89d725b59ee155b3f8be9d43780418b2ef7c3d348e6e393f71f42a0
6ed60a13554666cc3189658e39d80445f02dd7766f222f98534fd375e30ce97b
c82d3546a604b06dc2b7fc810364a976acb2a1046e19d12110fb5772dae690a5
8f6b5f4825a24ce6434ffa1714948b0446273cf21e89b91bf0058fd479b21d28
8aa75fba9ba6816cb369a972df75ede8245d94d176c335451f427374e108607a
88b1d6dfb749136e753ef211a4542adfe0f60e636f44eedfa1593b6a305a6830
660b49db66697858effa367d89ffdba72cf1597eb0e3c8a29d9ca97b99734c07
d04f22f68e83834dd3a62408e87ac050bc1fd0ad5df078c203f41a1ca96e9c82
27aaca0d5fb7efecb8b940ef4a7b9bf621031da90217cc3aa918f22babf9a761

927e7e925ce806e54936e31c8b3ddd18ffce40aa943d721e0e247c4c550ac54a
10fe7dc5da99cb200ac87a7c9e3141f9d2192e948032e5626bee7cd6f839f11d
eeaf0acc08318f0ac33a91492e4e6a54b6fd22e66bcbd769002e1164c67a7dc6
7345cf757c920b9ef9bcaaff82add6ff188827f442dc0eeee4145a576c1d837a
c9ae5c48e3f10a2a10d79fee0f6b9c2939b689b99ce51615bdef816bf0c1c1ee
72179ea91395cc99c030998885654e81186cf4461739ee1950a7a7d38f0c14ca
09e303e5bff760d22a19f870963f2caed2085ec402b617572ace7604acef8fce
fd0814b9ae83cf0e938efbbddab9a03811ee200bea3b92dc13e1722ab6ce2f21
17b68a6de7e964a6de915dbf26455e3685e5bd4c15d88b1a3aab61af59f5b1b6
eccc80ec056c711a86045324a6672804cd41377b4787b92521ac0c63c4dda3f9
49c1661f9f7a30b46a695633720572c5a644bcb913ff78bf87b1590637ff9fb1
c79a409842a83f2e2348406ee645b7608529bb6c3f81197ba311f8d932d09ffd
fe19a32b0ec4190e4445d1fcfa0949ac0385b3135b57d28d58d4ea98cb632841
dbea421f47b5f28202e3e45e74aafca5a18cfa37ee223cced5b4096ce9108067
3c97e3ea52da050ee8226bfec42b6a4b671aa6c38d0a16c630add523cdaf1bc0
3ea993da057a7fc1915045ffd1a5e49c77cd76e9cec9e0f2c19122b9965aaf6e
80c2e1aeda7e117028b7d3c519d1b5e1a911ef1b2d32dfa8411cdacd02eb239f
8af726da8d2c7d34188405f88f3f5e76803c8133d18bd5ceba8598b6a71c34f6
7b8f1a0f88301d761abed052b6cdc8a03c8ecd63737ce973059473292084f3d4
2ad9e0e6211378fe66479242dcb49ce74259e5fc9ffa6caac99cdd2b9a210088
d5248e90adc8d876521093af82784b8729bf72fbba7e886c2cd39f6b234cbfc0
97c4bee91c006d71052f91675fcd6c8c7db91ef435a61b6228ac1be3900c7730
78f6620d03c9df87bde82ca2445da9567cc84f02439010591a096fa5d662fc69
c1f9c066d5cbadf24d706ca6ce52eaedf8166058782d70f6f708ecd1be89c03b
030b0f2f1b0160140d435fdabb0c3276f122db1cd71775885d68f3127acd2d3d
8c1fac60bf303fad9cb7890f44e7a2a677e8d64c7c0df675cd97a3c8bfeaea16
7d5d1c2ed9dc2abd02d18e53004274392973d83f4e9b1013b4ea2b42c79f2a8e
fd3dd6371448fd73b65e467c2fef4b706102a177d3e52ed70ae272e58d6b01c0
637b84c391530200644ac364643b2db03da85b1ce0b5d916431837aff8270044
61ca2a5ed0add4bcc33e691e4f6e7f055950466a8abcabb38628069a33f86cbd
644a88d243277e377b73eaef7937c9b648328b1a006190cbedd075e4bbeb8835
ea44bf4f22cf5d26461049c3d3f9ea11bc2a7530e6c130402970ff9889294f15
1120cfd12e95f255317459c71c9f743f335482bb193db4afaae5238ae2110357
ed60f735a919a322e0eb135d0940b1f4ac2fbbe1db67bad4a27a5b3c63f8d6a2
4b08c9b9ee0ccdafe880211d9ff92dd0b0be0f452b333b5051a775c2ca01ecde
2a797ebdac7312b3943a241cfd9f176ebd37cea997b5f4c0b0b517e55b5c8928
61248f63c3bd9892c03a8353362ad9d6fc7a4fbc9349a53a4d659c66b0b1dd04
942aaef6466148193a9260117b75b6075ad81b3225fa564e89832429ff1b0f13
b08654590b66a89d94127adaeb6bc11d0bb38a4ec3a7cc60fe8fe46a29a7c638
c7c6b6bba7ba34bef589a686c3b0eb829f2170205bdff38611faf5d9e382e61e
7f5edbd292d9fb88d497a4181ff175fb5df20af38b749d4a9a3aa345e485336d
c92978a9472b12861042ddcc58c143462260a4a1ba1b0aab48bdc6368a2404f4
ed1ce8ed8591cb44d55e2c3c0e0d43721365c33be6788c10a623b92f57fab08c
7d2f03a8d436aaaa6dfd5d6750fa788f4661bede2ad34a3b9c1d72732f4d4af9

```
c1966c1004a0ff46137b1fc1d0976e99bfaa9c20028cc211cd4ceddcf7542495
ca931d7a64c16b05cc8907d401a09daab9683f62027cbf32b5c0585ae5443b75
aa2ff8c7efe0e40a9e1ee9304d54a1cf0f18b40ba6acc0878b5e60773674ae36
f8ff4d17a07ef824f23849d4411d42f13c02f498d8c796d0d18340e63815f2ba
05e05f35659fbc399b36ba24a214350af7fe23633ccaba1501de559970bb8830
81c9b174569c6700dd61e4df01881ac8ff9deb655c8f97dd4ffc6f773677bb1b
d6b5588cc757ab593ac31f55b106c58af4655faa9ca29082cd3ef04bb7da9b0d
ba2992b2fe0f5ff029a2f5564f54d22c5ba1add6e40cbdcb961544f19ff1c960
22d6a7241c0c3447f465eec620280dd892c11e3061ae56b9415380d3822fabc8
490361159e507290deb9fdae40f016203d152bd52f657234a3f73a3f77cd1ec7
6f6ebf54afe9009ad678fefc955ed948f48ff123efefe5df43b44d77f7c7ef05
08e2148167d7ac0bbbce6e0bc60815f37e29d0294bf28d75289fa21d0eec1a94
8d9b27f19f8353f1e60b4767034335be49278c9ed30b4b9e6f148b0b3ebd0fd2
41d988f7ca1e2d497a47795696d0e36fae34fb0957a303f6465e8ac951ccb3aa
6ad3c95a28a80b0da79dca7e41b1f3ce356f82caaa65737b20327b106d7c6390
fe020fad14b609a09c2914e6173b5e576360c955295cc89168ba84515e992f8b
241d85e9725d1b7941fbdcbb149f83bc533fed3ba86448858c3aa8a4f453bd12
c964b238b7a17947a7a236d95baea6a8b5bd59463d5df2a8d6c2a3c77c255761
de263e5ad81bb5e2be7d57c7e201fe172108d987562a98897736d8c9235661a2
b33c47e31d50d47631094546df901e96e8a9672df9f4bf53662683b12aa4c41e
841b66e71b363a181435f83ee197c83c0176837cd87953236f4fc839279549d7
f4bdb5409052e0a5b7598def8ea7c95e0411b1acb08aecc903770df83fa81643
43c2d27c6745596a99f3a2749029d705a57674ec695ba37ec5118ce4a5366c58
03cf69e25416b174614bb46f3182169d8bb85c17988c496b215988b8264a8903
dc83806bfbe5f7b940e1c442fa7558f3e060388cd2c7640e4cd6fe2f5b2a5ecb
a141bc729d7a2ed71e959ae1ff248e8e18cb47ecaf4300bf67d5d09681d174e1
3c02d100ded37943b6ee89eebedb00e9860a21a5affc087c4ee72a4c1db91adb
fa2793fb83110466b25ed53cd8da070486fcb0383a2cb24ca2485168d1107862
95eae63247e46dbca8faa5df5dbedf32cb8aa40b35893548c4c878a43fc82b53
d5272ecb6c1d5eb7a190ffcee0f39e493f6c7020bcd1680f6ca92c2a05f4c00e
6d466b5d1b3ddda411c60ceb3db50a33c987b1f30d96d9662e2d7d2eb1757cd8
235e596f3450cd1b3cc783bb20471b3fc146d5f24bf9fd1f3d282e5a8bc76e8e
41dc3dfd217b301809afcf5098c32280bc2857ca9c693f3053a465687281237a
```

Appendix C

Contents of Enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `bota` ..... BOTA implementation sources and documentation
  - `fet` ..... FET implementation sources and documentation
  - `notebooks` ..... the directory of ipynb notebooks
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `data` ..... custom dataset
  - `captures` ..... raw captures
  - `flows` ..... extracted flows
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format