Learning Automata for Network Behaviour Analysis

Pellegrino, G.

DOI
10.4233/uuid:6ea4a13b-ff8e-4c4b-866d-168cdccd880a

Publication date
2023

Document Version
Final published version

Citation (APA)
Pellegrino, G. (2023). *Learning Automata for Network Behaviour Analysis*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:6ea4a13b-ff8e-4c4b-866d-168cdccd880a

LEARNING AUTOMATA FOR NETWORK BEHAVIOUR ANALYSIS

LEARNING AUTOMATA FOR NETWORK BEHAVIOUR ANALYSIS

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus, prof. dr. ir. T. H. J. J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Friday 8 December 2023 at 10:00 o'clock

by

Gaetano PELLEGRINO

Master of Science in Computer Science, University of Naples Federico II, Italy
born in Naples, Italy

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. ir. J. van den Berg	Delft University of Technology, promotor
Dr. ir. S. E. Verwer	Delft University of Technology, promotor
Independent members:	
Prof. dr. M. M. de Weerdt	Delft University of Technology
Prof. dr. F. Ouardi	Mohammed V University in Rabat, Morocco
Prof. dr. G. Smaragdakis	Delft University of Technology
Prof. dr. F. W. Vaandrager	Radboud University in Nijmegen
Dr. S. Garcia	Czech Technical University in Prague, Czech Republic

This work is part of the research programme on Learning Extended State Machines for Malware Analysis (LEMMA), which was financed by the Netherlands Organization for Scientific Research (NWO).

Keywords: Automata Inference, Intrusion Detection, Passive Learning, Behavioral Fingerprinting

Printed by: ProefschriftMaken

Front & Back: Designed by Francesca Avitabile. The background image was AI-generated with Canva

ISBN 978-94-6469-686-8

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

To my father Giuseppe,
who passed away before I completed my thesis,
and to my daughter Giulia, who was born in the meantime.

A mio padre Giuseppe,
che mi ha lasciato prima che finissi questa tesi,
e a mia figlia Giulia che è nata nel frattempo.

CONTENTS

List of Figures

1

INTRODUCTION

1

It has been reported that the per-annum number of cybersecurity attacks has grown four-fold since 2019 [1, 2]. High as it is, this number is an underestimate because it considers only "interactive" attacks: hands-on-keyboard attacks carried out by people. The number does not account for attacks carried out through software tools. Many factors have contributed to the consolidation of this trend:

- The COVID-19 pandemic caused an increase in the attack surface. Many companies had to adopt remote working solutions quickly and often without due attention to security. The exposure of vulnerable teleworking infrastructures to cyberattacks [3, 4] has increased, as has the use of personal devices not controlled or managed by companies. According to some observers, these changes are irreversible and will continue to affect cybersecurity even after the pandemic ends [5, 6].
- Cyberspace has become an increasingly important dimension of political conflict. With the growing dependence of critical infrastructure and economic systems on digital technologies, cyberspace has provided a global-scale battlefield. The geopolitical impact of cyberattacks—especially those used for intelligence, sabotage, and propaganda—has been growing. One example of this was seen during the early months of the military escalation in the Russo-Ukrainian conflict, where cyberspace was used by both parties to spread propaganda and fake news [7]. Another example is highlighted by the cryptocurrency revenues generated by North Korea through cyber operations aimed at attacking investment firms and centralised exchanges [8].
- Ransomware attacks, in which cybercriminals encrypt a victim's data and demand ransom to restore access, have become increasingly common and sophisticated [9]. Many of these attacks now use double-extortion tactics, where the attackers not only encrypt the data but also steal and threaten to publicly release sensitive information if a ransom is not paid [10].
- Supply-chain attacks, in which cybercriminals compromise a third-party vendor or supplier to gain access to a target organisation, have become more frequent [11]. These attacks are particularly effective because the vendor or supplier often has less robust security than the target organisation.
- Cybercriminals use AI- and ML-based techniques to create convincing fake videos and images that can be used to spread misinformation or impersonate individuals [12].
- As more organisations adopt cloud-based solutions, more cybercriminals focus on attacks targeting these services [13]. These attacks often involve compromising a user's credentials or using malware to access data stored in the cloud.

The security controls implemented by many organisations have yet to keep up with the ever-increasing number and sophistication [13] of cyberattacks. As reported by a global cybersecurity services provider [14], only 26% of the cyberattacks that occurred in 2020 were promptly detected by the security controls in place; only 33% of those attacks were automatically blocked as soon as they were detected. Only 9% of cyberattacks triggered

security alerts, and as many as 53% passed unnoticed. In 2021, the median dwell time, which is the time between the start of a cyber-intrusion and its detection, was 21 days: a period considered long enough to gather relevant intelligence and perpetrate a nefarious activity [15].

It is difficult to quantify the damages caused by those security incidents. Reports published in 2021 [16, 17] estimate threat actors' profits to range from 123 million to 350 million U.S. dollars in 2020. However, those estimates are limited to ransomware and represent a lower bound to the actual, unknown quantification since they exclude possible impacts of reputational damages, various forms of collateral loss, and untracked ransom payments.

The lack of intelligence is one of the leading causes of a failed reconnaissance of a malicious approach [14]. Adopting Network Intrusion-detection systems (NIDSs) is one possible mitigation to this problem [18]. NIDSs are software systems that monitor networks for malicious activity. NIDS can be classified according to the detection methodology: knowledge-based (K-NIDSs) or behavior-based (B-NIDSs).

K-NIDSs[1] [19–21] rely on known patterns of malicious activity and determine whether these patterns match the captured network traffic. The knowledge base used for K-NIDS is typically composed of detection rules [22–26] or string patterns [27]. K-NIDSs are usually accurate in detecting known behaviours. Also, they tend to produce interpretable results because the detection engine is generally transparent to the client. As discussed later, interpretability is a desirable property enabling actionable intelligence. Furthermore, interpretability makes any system more trustable. However, K-NIDSs are less effective in detecting unknown attacks: the attacks where intrusion attempts are not matched by any pattern in the knowledge base. Another challenge with K-NIDS is updating the knowledge base with the ever-changing threat landscape. On the one hand, the knowledge base may become obsolete quickly because new attacks are discovered often. On the other hand, obtaining new patterns can be costly because they require technical know-how. Whether the knowledge base is created "in-house" by an internal team or acquired from trusted external sources, making and maintaining a knowledge base can incur substantial costs for an organisation.

B-NIDSs build a model that describes the normal behaviours in the network; later, they use this model to detect unusual behaviours. B-NIDSs differ from each other by the model they adopt. Some B-NIDS adopt statistical models [23, 26–29] and others rely on machine learning [30–39] or deep learning [31, 33–36, 40–54]. B-NIDSs can detect unknown threats if those threats cause the occurrence of anomalous behaviors. Furthermore, B-NIDSs do not require any knowledge base of malicious patterns. However, B-NIDSs tend to suffer from false alarms, i.e., harmless behaviours erroneously considered malicious only because they are uncommon. Furthermore, B-NIDS based on deep learning produces uninterpretable results. Deep-learning models are deemed opaque:

[1] K-NIDSs are also known as signature-based NIDSs or misuse-detection NIDS. B-NIDSs are often called anomaly detection NIDS.

an unsavvy user might need help understanding the remote causes of behaviour that these models believe is anomalous. Recently developed B-NIDSs adopt a structured detection methodology composed of a combination of various algorithms [30–32, 34, 36–39, 45, 46, 48]; there also exist B-NIDSs adopting deep-learning models, such as autoencoders [31, 33, 35, 44, 46, 50, 52, 54]. In both combined and deep-learning based NIDSs, it might be difficult to determine why a pattern observed in a network capture has led to a detection, and this has a detrimental effect on the interpretability of the outcomes.

In summary, we need a better means of detecting cyberattacks against our systems in terms of quality and interpretability. Only by stopping attacks at the earliest stages will we be able to reduce the dwell time of the intrusions in our networks.

This introductory chapter is structured in three sections. Section 1.1 states the goal for the research discussed in this thesis. Section 1.2 provides a high-level overview of the principal contributions. Finally, Section 1.3 outlines the structure for the remainder of the dissertation.

1.1. RESEARCH GOAL

This thesis aims to investigate whether it is possible to create intrusion-detection systems that do not rely on rules created by domain experts, are not affected by the problem of ruleset obsolescence, are capable of detecting unknown threats, and are capable of producing interpretable results. The main result of our research is a methodology that allows the use of NIDSs based on libraries composed of computational models called automata. In the proposed methodology, each automaton can express multiple behaviours observed in a network capture.

There are a few reasons automata might be considered a good fit as models for intrusion-detection systems:

- Automata are particularly suitable for describing software behaviour during its execution. They have a decades-long history of being used to describe software dynamics [55, 56] in software design. We define the behaviour of software as the changes in internal states of the software and the expected outputs caused by sequences of input stimuli.

  This motivates our interest in automata. If automata have been used for years to describe the behaviour of software during design, why not keep using them to describe the behaviour at later stages of the software lifecycle?

- Automata are interpretable. Because automata are mathematical models, each component of an automaton structure, regardless of what that automaton represents, has a formally defined semantic. Furthermore, automata enable a human to track how an input to an automaton leads to the output. That is manually accomplishable if the automata are simple or with tool support in case of more complex instances.

This allows us to interpret the outcomes of an intrusion-detection system. In many use cases, the raw outcome of a system is not enough; an explanation of the reasons or causes for that outcome is requested to let human consumers trust the system and account for possible errors. The more critical the use case, the more it requires an explanation.

- Automata are models that can be learned from network captures automatically. This is possible thanks to many years of research in the grammar inference field, which produced several automata-learning algorithms. Section 1.2.4 introduces the topic of automata learning, and Chapter 2 discusses it in detail and provides instances of automata-learning algorithms.

  This alleviates the problems related to maintaining knowledge bases. Automata are automatically created from network captures, with little human effort and, more importantly, with little domain-specific expertise.

- Automata are models that describe not only the patterns observed in network traffic but also the potentially malicious patterns that have not yet been observed. Automata-learning algorithms make this possible because they can abstract the data provided to them and infer generic models. Chapter 7 discusses this capability in detail and illustrates it with a dedicated experiment.

  This makes the proposed methodology less prone to false-negatives issues that often characterise the K-NIDSs. An intrusion-detection system that implements the proposed methodology can detect attacks not present in any capture used for learning the automata in a library.

1.2. USING AUTOMATA TO DETECT NETWORK INTRUSIONS

The use of automata to describe attacks or intrusions is not a novelty. For example, Kumar and Spafford [57] describe a knowledge-based and host-based intrusion-detection system that relies on a library of known attack patterns modelled as coloured Petri automata [58]. However, those automata were designed by domain experts, which posed the challenge of maintaining the library. Furthermore, learning Petri nets automatically is a difficult problem [59]. In [60], the authors describe a knowledge-based and host-based intrusion-detection system for UNIX systems called USTAT. USTAT rely on a library of attack patterns expressed by finite-state machines [61]. However, as for Kumar's and Spafford's systems, those state machines were designed by domain experts to depict possible compromises of the system.
The rest of the chapter expands on the concepts introduced so far and is organised as follows: Section 1.2.1 describes the motivation for the interest in network data. Section 1.2.2 discusses the benefits of behaviour-based detection. Section 1.2.3 presents the automata and explains why they represent a reasonable choice for describing behaviour patterns. Section 1.2.4 introduces some concepts about automata learning. Finally, Section 1.2.5 provides an overview of the proposed methodology.

1

1.2.1. FOCUS ON NETWORK DATA

Cyberattacks are commonly modeled by using the Cyber Kill Chain®: a sequence of phases each representing a typical phase of a cyberattack. According to the Lockheed Martin Corporation [62], which imported the concept from the military and adapted it for the cyber world, the phases of the Cyber Kill Chain® are Reconnaissance, Weaponization, Delivery, Exploitation, Installation, Command and Control, and Action on Objectives. During Reconnaissance, the threat actor selects targets by researching and collecting data that can be used in later phases. After collecting enough information, the threat actor obtains tools for accessing and controlling the targets during Weaponization. We refer to those tools as "payload": a piece of software purchased or developed by the threat actor and, once installed, gives the threat actor control of the target system. Having obtained the payload, the threat actor attempts to deliver it to the target; for example, a phishing email campaign was one of the most common means of delivery in 2021 [1, 16]. After shipping the payload to the target, the actor attempts to install the malicious code. This phase is referred to as Exploitation and is commonly accomplished by using psychological or social manipulation to trick a victim into opening a file attachment. During the Installation phase, the threat actor attempts to install a payload to establish the persistence of the malicious code on the infected system. Once installed, the payload may set up a communication channel from the infected system to the threat actor's infrastructure; this phase is known as Command and Control (C2). Finally, the Action on Objectives phase is a generic container for all actions an intruder might perform to achieve its end goal. Some examples of actions are data exfiltration, file encryption, lateral movement to other infected systems within the same network, and erasing evidence of the intrusion.

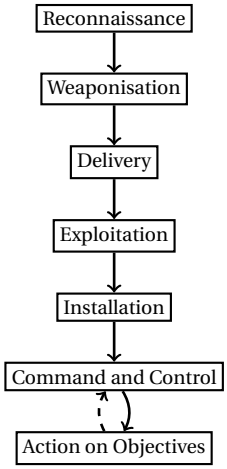


Figure 1.1: The cyber killchain is an established model for a cyber attack process [62].

Figure 1.1 summarises the Cyber Kill Chain®. Usually, the last two states might be repeated for several cycles. The return connection from Action on Objectives to Command and Control highlights the fact that in practice, the Cyber Kill Chain® does not always

proceed according to its definition [62]. A cyberattack might not traverse all the states of the chain. As an example, an insider attack[2] does not require a proper Reconnaissance phase if the threat actor already has knowledge of the target network and the authorisation to access the systems of interest. As another example, in some ransomware attacks, the actor does not need to establish a Command and Control connection with the opposing infrastructure.

In the Cyber Kill Chain®, at least four out of seven states require some form of interaction on a network. One of the most common reconnaissance actions is the port scans to gain insight into the network services active on the targeted system. Port scans are perpetrated by means of software tools[3] and remotely launched by the threat actors. The payload is usually delivered through a network interaction, either by email or by remote exploitation of some network-exposed service. Command and Control is yet another example of a phase where a network interaction is necessary because the threat actor is remote. This phase is particularly important with botnets, where a network of compromised systems is remotely controlled by an actor. Many of an actor's actions might suggest some sort of network interaction, such as lateral movement, where the intruders move from one infected host to another within the same network. In addition, recognising those malicious interactions by analysing network data is crucial for detection and post-hoc analyses, such as forensic investigations. NIDSs might trigger a detection in a very early phase of a cyberattack: in the reconnaissance phase, when the intrusion has not happened yet.

Network data might take many forms and granularities. The methodology described in this dissertation is based on Netflows. If a packet represents the atomic unit of information carried by a network, a Netflow [63–65] (or network flow, or just flow) is an aggregation of packets that share the same key. A Netflow key is usually composed of a 5-tuple that comprises the source address, source port, destination address, destination port, and protocol. All packets that share the same key in a specific timeframe are considered part of the same Netflow. A Netflow is the summary of specific features of the packets, and it comprises the duration, the size in bytes, and the number of packets. Netflows come with interesting properties, making them the data type of choice for several analytical tasks [66]. Even a single system can produce large amounts of packets in one day. The escalation of those amounts on small-to-medium networks makes it almost impossible for a software tool or a human to scan every packet — the packets are simply too numerous. However, because each flow is an aggregation of multiple packets and because processing a flow means processing all the packets it aggregates, Netflows scale better than packets on significant amounts of data. Nowadays, about 85% of the network interactions are encrypted [67]: that is, the content of packets is transformed to protect confidentiality. Most of the time, neither humans nor software tools can look into an encrypted packet's content. Even when a decryption key is provided, privacy, legal, ethical, and social considerations often thwart attempts to analyse the packet's contents. Because Netflows consider packets' metadata rather than content, flows are unaware of

[2] An insider attack is perpetrated in collaboration with at least one member of the targeted organisation.
[3] nmap (`https://nmap.org`) is one of the most popular port scanners.

encryption. Naturally, if compared to the content of an unencrypted packet, the information content of a Netflow is far more limited.
An interesting aspect of network data is that once cyberattacks occur, their tracks cannot be tampered with or erased, as happens for interactions with the operating system on the compromised endpoint. Indeed, once threat actors gain control of the targeted system, they might wipe off access logs and install rootkits to hide files or processes of interest to the operating system itself and to any possible endpoint security software. These and other actions hamper both detection and post-hoc analyses. On the contrary, once a network transmission is sent out on the wire, the data is hardly under the threat actor's control. This is even more evident in recently deployed and appropriately designed networks, where network data is gathered by sensors placed in strategic points and log aggregators installed on dedicated hardware located in controlled segments. In Messier's words, "the network cannot lie" [68].

1.2.2. BEHAVIOR AS A FEATURE

Indicators of Compromise (IoCs) are technical characteristics found in system or network artefacts that represent evidence of malicious activity. IoCs are used for detecting a possible cyberattack and confirming one that has already happened. As an example of IoC-based detection, antivirus software detects malware by matching file signatures found on the attacked system with known malware signatures included in the antivirus's dataset. Those malware signatures are often composed of IoCs that security researchers extract after analysing malware. Similarly, NIDS detect malicious activities by applying filtering rules to the network traffic. By matching a packet's content against known IoCs, the rules dictate when a packet must be considered malicious. As an example of IoC-based post-hoc analysis, IoCs are used to address the attribution problem: that is, the process of tracking and identifying the threat actor behind a cyberattack [69]. Attribution of cyberattacks is based on the idea that nothing a human makes can avoid personal expression, including the choice of Tactics, Techniques, and Procedures in cyberattacks (TTPs). That type of personal expression might be unique enough to support an attribution hypothesis. In attribution, threat actors are represented by intrusion sets: groups of IoCs consistently observed in different cyberattacks. Threat actors are mapped to possibly several intrusion sets.

The most common IoCs are hash values, IP addresses, domain names, network artefacts, host artefacts, tools, and TTPs.

- Hash values, or just hashes, are the output of the application of specific one-way functions (i.e., the hash functions) to the malware samples. Since just a single bit flip into the malware sample leads to a different hash, hash values are commonly used to univocally identify malware samples.

- IP addresses are numerical values uniquely identifying devices connected to a network via an IP internet protocol.

- Domain names are strings that uniquely identify web servers and other network-exposed resources. Domain names serve as human-friendly labels for IP addresses.

- Network artefacts (for example, URI[4] patterns) are observable features left by the threat actor on the target network.
- Host artefacts, as the name suggests, are the host-based counterparts of network artefacts. Therefore, host artefacts are observable features left by the threat actors interacting with the operating system. Host artefacts include special registry keys or folder names touched by threat actors.
- Tools are the software utilities threat actors use to achieve their goals. They can be malware of various forms, such as a backdoor or ransomware, or legitimate programs used with malicious intents, such as port scanners and penetration-testing frameworks.
- Finally, TTPs are definitions of the methods used by the threat actors. The definitions must be sufficiently generic to be abstracted from the actual tools but sufficiently specific to profile the threat actors.

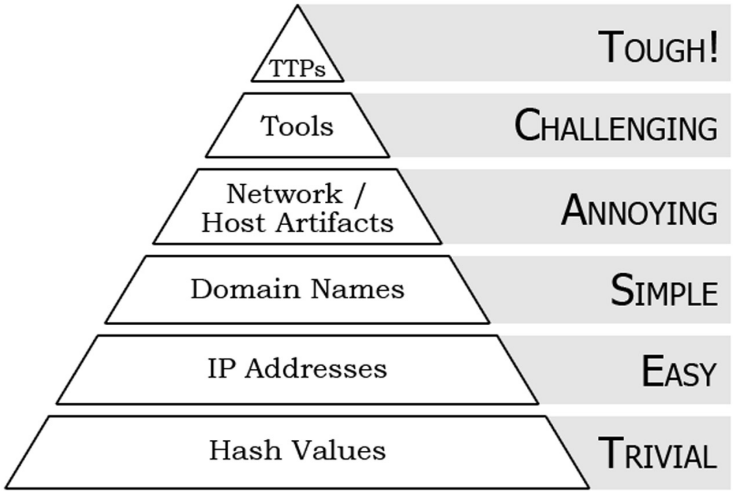


Figure 1.2: The Pyramid of Pain [70] shows the stability of each type of Indicator of Compromise (IoC).

Stability is a desirable property of IoCs and is defined here as the capability of IoCs to persist across multiple cyberattacks. The concept of IoC stability has been addressed in [70] with the so-called Pyramid of Pain (PoP), shown in Figure 1.2. In the PoP, stability is expressed in pain for the threat actor in changing a given type of IoC. The bottom layers of the pyramid refer to the most unstable IoCs because the threat actor can change them with little to no pain. On the contrary, the top layers of the PoP refer to the most stable IoCs because changing them would require much more effort from the threat actor. For example, though unique for each artefact, hash values are considered unstable because even a single-bit mutation in the input artefact leads to a completely different hash value. On the contrary, TTPs are the most stable type of IoCs because adopting

[4] Uniform Resource Identifier.

a different technique or procedure implies a fundamental drift in the threat actor's behaviour. Thus, changes in TTPs are supposed to be costly because they require the actor to learn new skills or hire professionals who possess them.

Although TTPs and tools indicators are the most stable, they cannot be automated. An IoC is considered automated when software tools can extract and process it. TTPs and tools are indicators related to decisions and choices made by human beings, and that is why they are so stable. They are out of the scope of discussions about software tools used to enhance detection. Unfortunately, the only IoC types available for software tools are also less stable. That is why keeping the K-NIDS knowledge base up to date is essential to effectively protecting systems and networks. Automated indicators are unstable because they are usually controlled by the threat actor. For example, in most cases, there is no need to recompile the source code to change the hash value of malware; the threat actor might only need to flip a bit in the binary. Similarly, even if the C2 IP address is hardcoded in the binary, the threat actor might only need to rebuild the binary to commit the update. If the malware adopts a C2 domain, then no update to the binaries will be required to change the IP address and keep the C2 channel alive.

The PoP, in its original formulation, does not mention an additional form of an indicator: the behaviour. Behaviour is a running software system feature consisting of any observable process interaction with the execution environment. We define behaviour as the expected outputs or as the observable changes of internal states of a system caused by a sequence of stimuli coming from the execution environment or from other software systems. Behaviour is a more stable indicator than hash values or IP addresses because it is a feature that the threat actor usually cannot control. Indeed, changing the behaviour of malware usually requires changing the source code so much that the malware appears to be completely different after the commit. For example, changing the way ransomware encrypts files on an infected system, in most cases, requires editing the codebase. A relatively recent trend in detection software development shows an increasing interest in software behaviour. The underlying idea is to detect malicious activities not by looking for matches with unstable IoCs but by looking at how malicious software interacts with the environment once it is executed. Indeed, increasingly, antivirus vendors integrate capabilities for detecting malicious software behaviour into their products [71–74]. Describing malicious software's behaviour is debatable because there are different ways to do it. The following section introduces the technique we consider the most reliable.

1.2.3. AUTOMATA AS MODELS OF SOFTWARE BEHAVIOR

There is no consensus on how to define the behaviour of malicious software. Vendors of security products tend to conceal both the source of behaviour-related information and, more importantly, the model of the behaviour their product relies on. In some cases, the behaviour of malware is thought of as a summary of IoCs collected after an intrusion into sandboxed environments is simulated[5]. VirusTotal, a well-known aggregator

[5] In the context of malware analysis, a sandbox is a controlled and isolated environment used to run malware for the sake of collecting IoCs and recording all its interactions with the operating system.

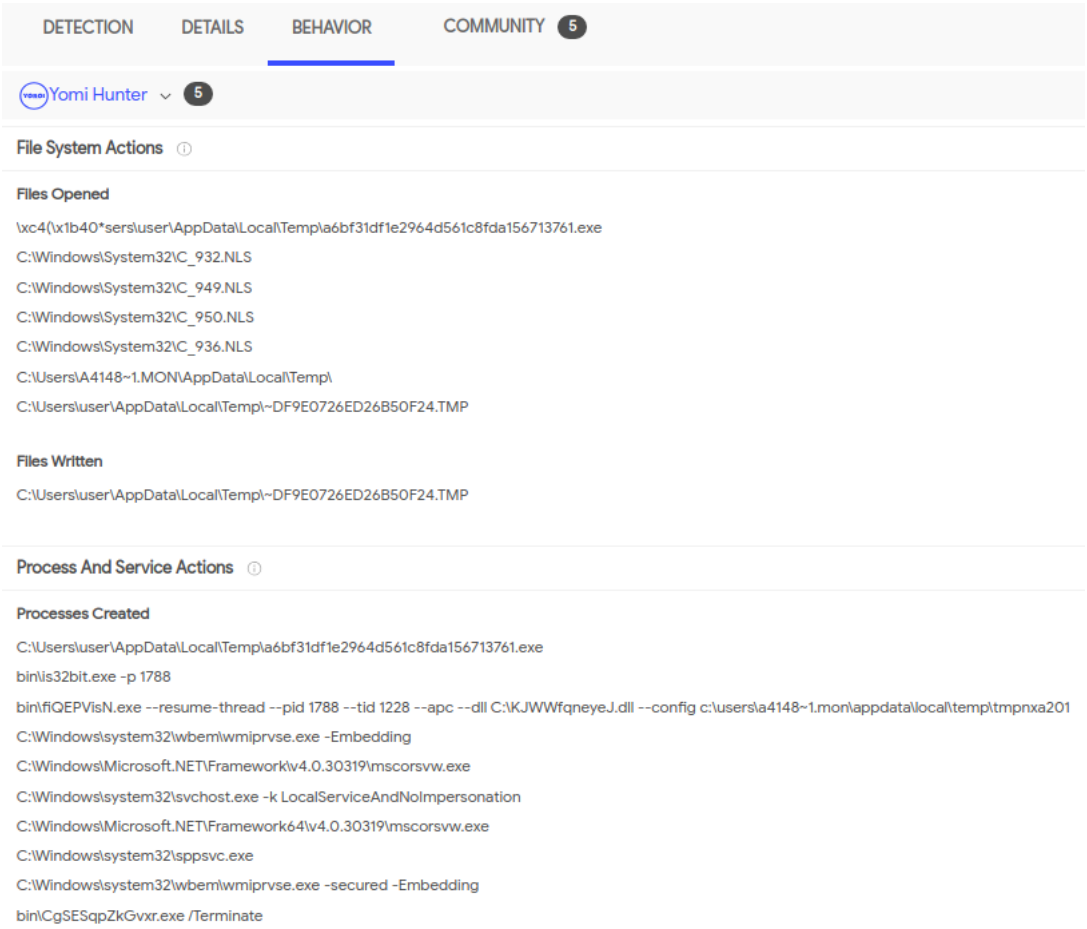


Figure 1.3: The behaviour tab on VirusTotal gives a flat description of malware behaviour. This is a partial view of the tab content.

of antiviruses, offers an example. It allows a client to submit a file for parallel scans operated by an array of different antiviruses[6]. The BEHAVIOR tab in VirusTotal, shown in Figure 1.3, consists of an enumeration of the IoCs gathered after a sample is run in one of several available sandboxes[7]. This behaviour concept cannot represent interaction dynamics because it shows the IP addresses contacted, the folders created in the file system, and other information. It does not provide information about, for example, creating a folder due to a successful first contact with the threat actor's remote infrastructure. In other words, this model's behaviour fails to represent relationships between the observed indicators. Furthermore, this way of expressing behaviour is still based heavily on IoCs, which, as discussed in the previous section, might be a somewhat unstable feature.

In this dissertation, the proposed methodology uses automata to describe the behaviour of malicious software. Automata are abstract machines composed of states and capable of processing input messages. At any moment, an automaton is in a state that reflects its reaction to some input message. In other words, during a computation, the automaton transitions from the current state to the next state, and the next state depends exclusively on the current state and the received input message. Figure 1.2 shows an example of an automaton and provides a high-level description of the possible behaviour of malware that allows to determine whether the threat actor is to be contacted via the C2 channel. The states of the automaton are represented with circles, and the transitions between the states are represented as arrows. Each transition links a source state to a destination state and is decorated with a symbol representing the input message triggering the transition at its occurrence. The source-less transition in the top-left state indicates the starting state: the state of the malware when installed. In this high-level behaviour model, the malware contacts the threat actor only if it has been launched and only during nighttime. Indeed, the C2 state is reached if and only if the automaton receives a "run" message, which triggers the transition from the starting state to the Running Silent state, and this message is followed by a "night" message, which triggers the transition from the Running Silent state to the C2 state. When a new day comes, the malware becomes silent: it is running but not communicating with the threat actor. Indeed, being in state C2, and after receiving the message "day", it transitions to Running Silent again.

There are several reasons for choosing automata to model the behaviour of software. As a first reason, automata have been used for decades to design legitimate software. For example, Unified Modeling Language (UML) [55] is probably the most common language for system design. To this end, UML comprises different types of standardised graphs, called diagrams, each providing a uniquely informative point of view of the software under specification. One of those diagrams is called statechart[8] and is included in the UML suite of diagrams to express the intended behaviour of the system at design time. UML state charts describe state machines, which are the most common types of automata[9]. Another example of automata's broad applicability in describing the behaviour of legiti-

[6] For more information, see `https://www.virustotal.com/gui/home/search`, section "How it Works".
[7] In this case, as shown at the top of the picture, it is the Yomi sandbox [75].
[8] Actually, UML statecharts were first defined by David Harel in [56], and that is why they are also called Harel statecharts
[9] The automaton in Figure 1.4 is an instance of a state machine

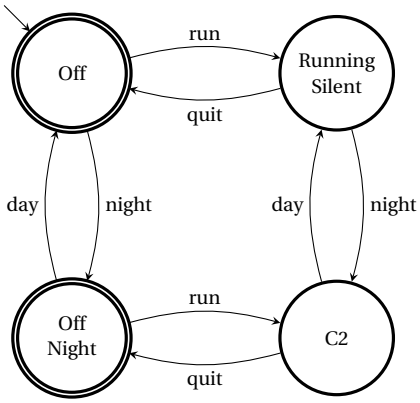


Figure 1.4: An automaton representing an over-simplified malware behaviour. It connects to the Command & Control server (C2) if and only if it gets executed and it's nighttime

mate software comes from specifications of communication protocols. Communication protocols (henceforth referred to as "protocols") are systems of rules that enable communication between two or more entities. TCP-IP is a suite of communication protocols that allow two or more computers to communicate on the internet[10]. A protocol's definition is usually composed of two parts:

- A static part defines the structure of each type of an allowed message. This is where message format is specified, such as the number of fields a message can have, the admissible values for each field, and what those values represent.
- A dynamic part defines how to build a conversation compliant with the protocol. This is where the communication rules get established, such as how to start a conversation, what to expect after a certain type of message is sent, and how to reply after a certain message is received. The dynamic part is represented by a form of automaton: a state machine. For example, Figure 1.5 shows the state machine defining the TCP protocol dynamics as they appear in the protocol specifications [76].

A second reason for choosing automata for modelling software behaviour is interpretability. Interpretability is a desired quality for a model because it enhances trust, i.e., the degree of comfort for a human to make decisions based on the model outputs. Trustability is achieved when it is possible to understand whether a model performs well and the inputs with which it performs well. Automata are interpretable according to all the meanings discussed in [77]. For example, automata are interpretable because they can be simulated, especially if the number of states is reasonably small. A model is simulable when a human can take the inputs with the model parameters and step through the calculations. As a further example, automata are interpretable because they are decomposable. A model is decomposable if each of its components is inherently explainable.

[10] TCP-IP is also known as the Internet protocol suite

1

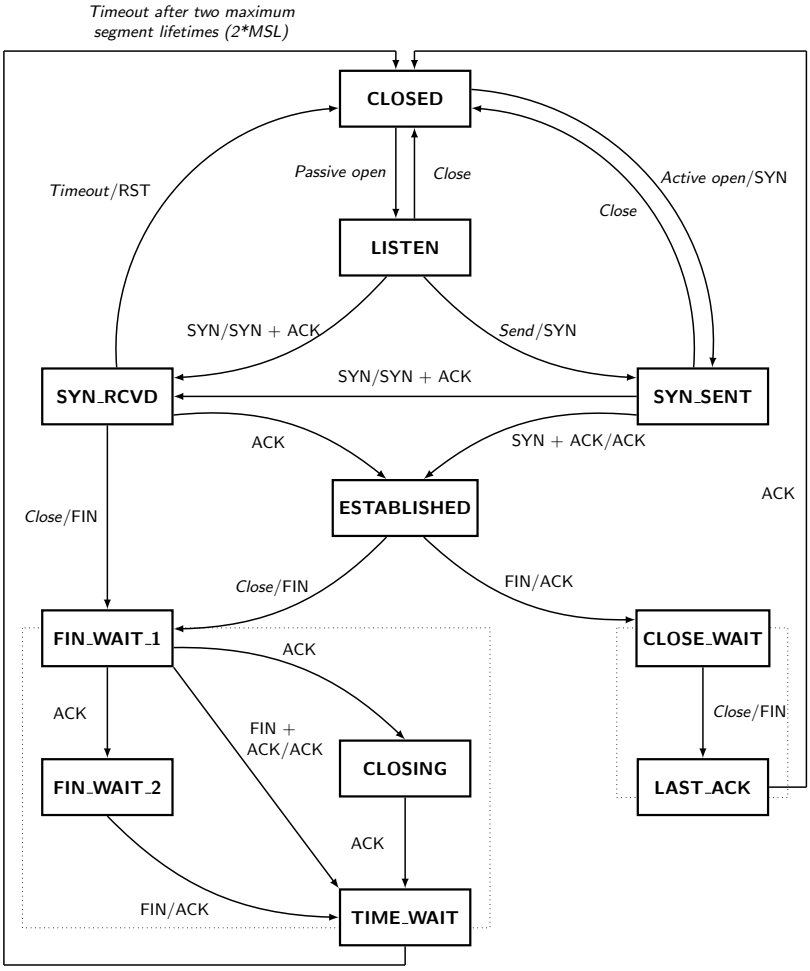


Figure 1.5: State machine of the TCP protocol. Credits: Ivan Griffin.

Indeed, as discussed in the case of the state machine shown in Figure 1.4, each automaton state and transition has its own meaning.

The last reason for choosing automata as models of software behaviour is that they can be learned from data. This is an essential property of a model suitable for the methodology proposed in this dissertation. Without this property, it would be challenging to devise a single automaton that describes network interactions starting from a Netflow capture. Inferring an automaton from data of any form can be a time-consuming, difficult task for a human being.

1.2.4. LEARNING AUTOMATA FROM DATA

Inferring automata directly from data — having some algorithm infer automaton with little to no human intervention — is a crucial part of this research. For this reason, the goal of this section is limited to introducing some concepts about training automata. Chapter 2 offers a much more comprehensive discussion by providing formal foundations and details of some state-of-the-art automata learning algorithms.

One way of introducing automata learning is by connecting this topic to a different, but probably less abstract, one. In ethology, scientists study animal behaviour: that is, they apply the scientific method to determine the causes of some interesting behaviour and, eventually, expand knowledge about a specific species. One of the early stages of ethological research consists of defining the behaviours of interest and, ultimately, observing the animal in the wild. While observing the targeted animal, the ethologist records all observed behaviours over time. One of the most common tools for this is the ethogram[11] a research artefact made of two essential parts:

- First, there is a catalogue enumerating an animal's behaviours or actions. This enumeration must be as complete as possible to provide meaningful answers to the ethologist's questions. For example, if the ethologist wants to study the hunting behaviour of wolves, the expected ethogram catalogue could be similar to that shown in Table 1.1, which focuses on the hunting-related aspects and neglects the rest.
- Second, there is a so-called data-collection part of the ethogram. This part records the occurrence of behaviours observed in the animal over time. These are sequences of behaviours mentioned in the catalogue in the first part of the ethogram. They provide the basis for any later hypothesis on the animals' behaviour.

Usually, an ethologist approaching the study of a species starts with an initial hypothesis that answers a specific research question. Observing that species in the wild may confirm or deny this initial hypothesis. Indeed, if a contradiction is found during the data-collection process, then the ethologist might decide to formulate a new hypothesis that reflects the observations. This process repeats until the ethologist considers the current hypothesis as answering the research question. What we have just described can be considered as an instantiation of the empirical cycle [80].

[11] For more information, see an introduction to the behaviour-measurement subject [78].

Table 1.1: example of an ethogram for the behaviour of large carnivores (e.g. wolves) hunting ungulate prey, taken from [79].

Foraging state	**Definition**
Search	Traveling without fixating on and moving toward prey
Approach	Fixating on and travelling toward prey
Watch	Fixating on prey while not traveling (e.g., standing, sitting, or crouching)
Attack-group	Running after a fleeing group or lunging at a standing group while glancing about at different group
Attack-individual	Running after or lunging at a solitary individual or a single member of a group while ignoring all other group members
Capture	Biting and restraining prey

The process of gaining an understanding of animal behaviour from direct observation is similar to the process of learning automata from data. I f the research into behaviour starts with a researcher defining the ethogram catalogue by filling it with interesting and repeatable behaviours, then part of an automaton's definition comprises a set of input messages the automaton should process. Because the input messages are called symbols, that set of input messages is called the alphabet. For example, the alphabet for the automaton in Figure 1.4 is composed of the symbols day, night, run, and quit. Notice that it is possible to give a definition for each of those symbols because they were items in an ethogram catalogue, shown in Table 1.2.

By continuing with the similarities, if the behavioural research relies on sequences of observed behaviours exhibited by the animal over time, the automata learning process depends on words produced by the targeted unknown automaton. The observed behaviours of the animal are those inserted into the ethogram catalogue, and the symbols that make up the words are those enumerated in the automaton alphabet.

Table 1.2: Alphabet for the automaton in Figure 1.4 with a possible explanation for each symbol.

Symbol	**Definition**
Night	It is night-time in the timezone of the victim (e.g. by checking the operative system clock)
Day	It is day-time in the timezone of the victim (e.g. by checking the operative system clock)
Run	Starting the execution
Quit	Terminating the execution

The last similarity lies in the learning process itself. If the ethologist reformulates and adapts the hypothesis according to the collected observations, the same happens for many learning algorithms. Indeed, when learning automata, the hypothesised model

is constantly updated to keep consistency with the observed words. Those updates are called mind changes. A learning algorithm is said to converge in the limit when, after observing many words, it will reach a point where there will be no mind changes. The convergence to the limit is a crucial property of learning algorithms; without it, there would be no certainty that an algorithm would learn the correct automaton. This is similar to studying animal behaviour because the ethologist will stop the process when the hypothesis answers the research question.

The final consideration of automata learning concerns the importance of the automaton structure, which consists of the states and transitions between them. The structure is essential because it makes automata interpretable by lending transparency, trust, ability to simulate, and informativeness (see Section 1.2.3). In machine learning, the learning of some models is reduced to a problem of estimating the parameters because domain experts define the structure. A typical example is the case of neural networks, where the network architecture is fixed, and the learning algorithm is limited to estimating the model weights. All states of algorithms used in automata learning, especially those discussed in the following chapter, aim to learn the possible model parameters and, more importantly, the structure of the target automaton. Naturally, that aspect complicates the learning problem.

1.2.5. THE PROPOSED METHODOLOGY

The main contribution of this research consists of a methodology for learning automata from Netflow captures and, through them, determining the suspiciousness of host behaviours. Section 1.2.2 expressed the reasons for a growing interest in behavioural features for both detection and post-hoc analysis tasks. Furthermore, Section 1.2.1 highlighted the benefits of working with network data in general and with Netflows in particular. A host behaviour might be considered the aggregation of all its interactions during a specific period. Those interactions happen according to the communication protocols implemented in some layer of the suite of TCP-IP protocols. To this extent, the behaviour of a host network might be considered as determined by those software implementations. As such, the behaviour of a host network can be described with automata.

The proposed methodology adopts a host-level granularity: it aims to inspect behaviours per host, even if not limited to that. In principle, either a finer per-connection granularity or a coarser per-network granularity might be adopted. However, the decision to centre the methodology on hosts has been made after considering that host-based judgments are more suitable for most detection cases. Knowing which hosts are suspicious within a possibly vast Network Under Observation (NUO) is more valuable than classifying each Netflow passing through the observation points. The possible remediations are also host-based because a host might get isolated from the NUO during the investigations and eventually reverted into a safe, pristine state in the case of a confirmed infection.

The proposed methodology consists of two distinct phases: training and testing.

The training phase, shown in Figure 1.6, assumes the availability of one or more Netflow

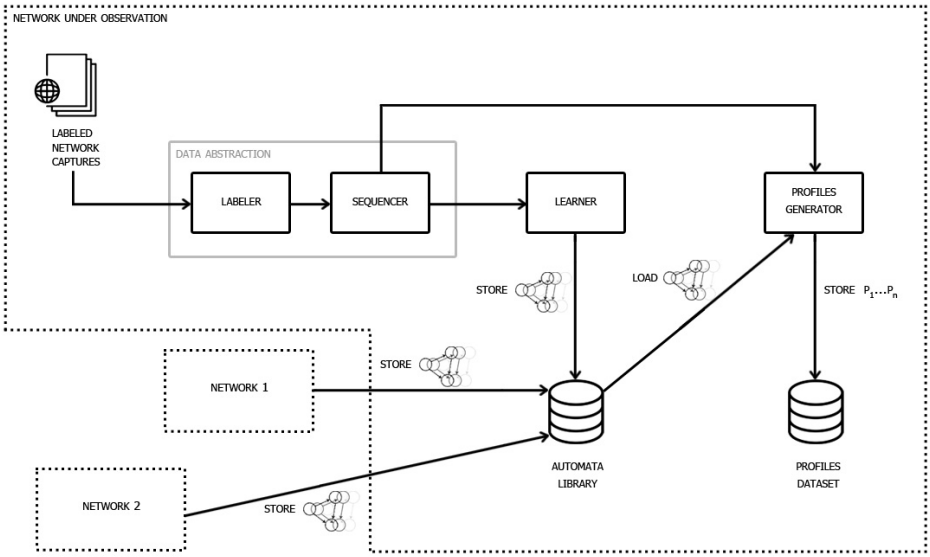


Figure 1.6: Training phase of the proposed methodology.

captures called training captures. Each training capture contains Netflows produced by a host that belongs to the NUO. Training captures have the characteristic of being labelled: that is, it is known whether they were collected from an infected host or from a presumably safe host. That scenario is considered reasonable because captures of malicious activity might get collected from hosts involved in security incidents, and captures containing legit traffic might get collected from just reset hosts.

As discussed in Section 1.2.4, learning an automaton requires both an alphabet of symbols and a training set of strings: words composed of symbols taken from the alphabet. Therefore, one of the first steps of the training phase is to generate an alphabet and a training set out of each training capture. It is called the data abstraction step and is implemented via two separate components: the Labeler and Sequencer.

The Labeler maps each Netflow to a symbolic counterpart. There are at least two strategies for achieving this goal, and both are discussed in Chapter 3. As an example, it is possible to label Netflows according to the duration and the amount of bytes transmitted. This criterion could lead to the following symbols: "Slow and Light" flows (SL), "Slow and Heavy" flows (SH), "Fast and Light" flows (FL), and "Fast and Heavy" flows (FH). Depending on the training capture, the Labeler could map a Netflow as fast if its duration is less than 1.2 milliseconds and as slow otherwise. Similarly, the Labeler could map a Netflow as light if the number of bytes transmitted is less than 124 and as heavy otherwise. Consequently, a Netflow lasting 7 milliseconds and carrying 30 bytes would be labelled with the symbol SL. The Labeler maps a stream of Netflows, contained in a capture, to a stream of symbols.

The Sequencer transforms the output of the Labeler into a training set of words. As for the Labeler, different strategies (see Chapter 4) are used to obtain words out of a stream. Once the data abstraction step is over, each training capture is translated into a training set ready to feed an automata-learning algorithm.

The learning component, the Learner, strongly depends on the type of automaton adopted as the target model. In Chapter 7, the model of choice consists of Probabilistic Deterministic Automata (PDA), which are learned with the ALERGIA algorithm; both PDA and ALERGIA are discussed in Chapter 2. In Chapter 6, the model of choice is Probabilistic Deterministic Real-Time Automata (PDRTAs), which are learned with the RTI+ algorithm. PDRTAs are discussed in Chapter 2, and RTI+ is discussed in Chapter 6 as it represents an enhanced version of the RTI algorithm introduced in Chapter 2.

Each automaton learned from a training capture is stored in a so-called Automata Library, which might contain the automata learned in the NUO and the automata learned in other networks and eventually imported. That is made possible by the data-abstraction procedure, which makes the learned models sufficiently generic to be applied in networks other than the one where they were identified.

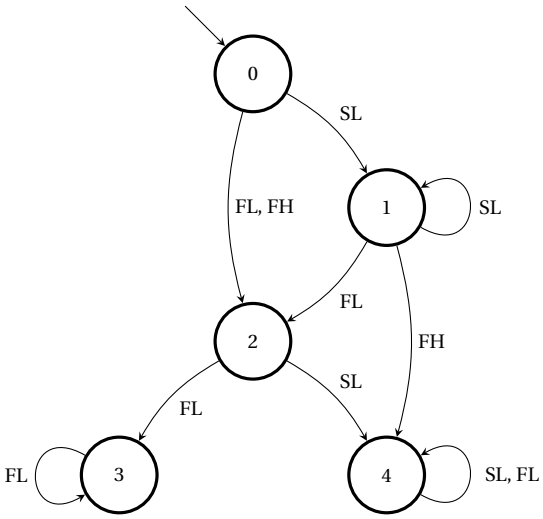


Figure 1.7: Example of automaton learned from a Netflows capture.

Automata are used to generate network-specific behavioural features called symptoms. A symptom is a couple (q, α) where q is the automaton state reached after processing symbol α. Given an automaton and a word, symptoms can be generated by letting the automaton process the word. As an example, consider the automaton in Figure 1.7 as having the alphabet $\{SL, FL, SH, FH\}$. By processing the word SL, SL, FH, the automaton

produces the following symptoms:

$$(1, SL), (1, SL), (4, FH).$$

Initially, the automaton is in state 0: the initial state (pointed to by a source-less transition). The first symbol of the word, SL, activates the transition from state 0 to state 1; therefore, symptom $(1, SL)$ is generated. The second symbol of the word, SL, activates the transition from state 1 to state 1 itself and, therefore, symptom $(1, SL)$ is generated again. The process is reiterated for each symbol in a word and for each word in the training set. All symptoms generated from the same capture form a multiset called a profile. For example, after obtaining the set of words {[SL,SL,FH], [SL,SL]}, automaton generates the profile shown in Figure 1.7:

$$\{(1, SL) \times 4, (4, FH) \times 1\}.$$

All profiles generated for each training capture are stored in a Profiles Dataset collection during the training phase. If the training capture was collected from an infected host, the corresponding profile is called an infection profile.

The testing phase, shown in Figure 1.8, assumes the availability of one or more Netflow captures obtained from some host under observation and called testing captures. In contrast to the captures produced in the training phase, the captures are not labelled because the testing phase aims to assess whether a candidate host exposes suspicious behaviour. The first steps of the testing phase are functionally identical to those of the training phase. For each model in the Automata Library, a testing capture is abstracted to a dataset of words for that model. The components involved, namely the Labeler and the Sequencer, are the same as those of the training phase. The words are provided to each automaton to generate a profile by following the procedure described earlier. If any of those profiles matches a known infection profile, an alert is generated for the host that has produced the testing capture. At least two strategies are used to assess the similarity of the profiles, and both are discussed in Chapter 7 and implemented in the Profile Matcher component.

1.3. OUTLINE OF THE THESIS

This section summarises the content of the remainder of this thesis. The dissertation is divided into three parts.

The first part includes this introductory chapter and Chapter 2, which provides all the automata-related concepts and learning required to understand the other parts.

The second part addresses some of the issues relevant to learning automata from telemetry data, and it includes Chapter 3, Chapter 4, and Chapter 5. More precisely, Chapter 3 focuses on the problem of how to translate raw data into symbols. It introduces an innovative algorithm that integrates the Labeller and Learner components when the input data is sortable. Chapter 4 describes how to obtain words from a stream of symbols. This task is carried out by the Sequencer component within the proposed methodology dia-

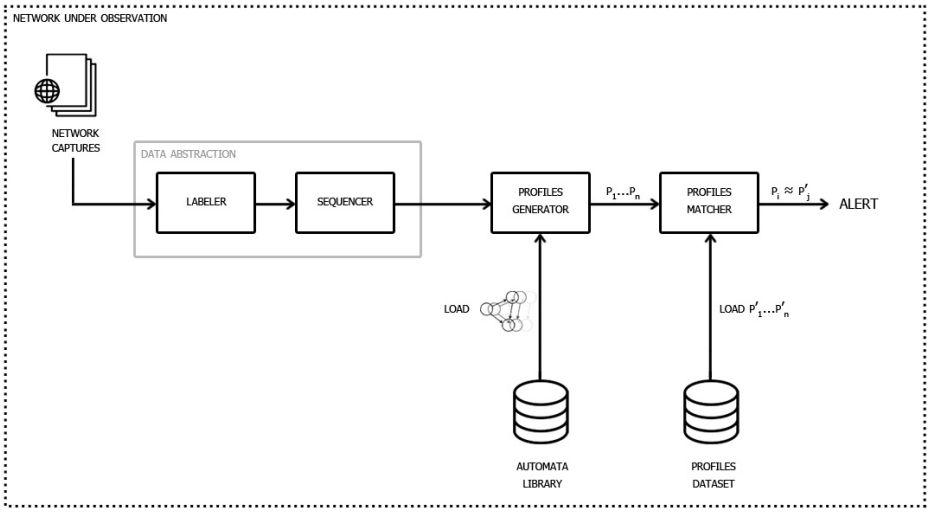


Figure 1.8: Blueprint of the testing phase of the proposed methodology.

gram. The chapter shows different alternatives and offers an innovative solution. Chapter 5 is focused on the problem of how to gather a limited yet relevant amount of data to learn automata from streams. It introduces a method for estimating the completeness of the collected data.

The third part presents two different applications of the methodology described in Section 1.2.5 and includes Chapter 7 and Chapter 6. In Chapter 6, we propose Probabilistic Deterministic Real-Time Automaton (PDRTA) as the automaton type of choice. Different to other automata, PDRTAs embed the notion of time. Due to the adoption of PDRTAs, the Learner component implements the RTI+ algorithm. In Chapter 7, the automaton type of choice is the Probabilistic Deterministic Automaton (PDA), and the Learner implements the ALERGIA algorithm.

The dissertation ends with Chapter 8, which outlines the main findings and proposes some ideas for future work.

2

Background

This chapter[1] introduces all the concepts which are required to understand the rest of the dissertation. Furthermore, it presents some of the most common models and the state of the art learning algorithms. Although this dissertation focuses on particular models and one learning framework, the chapter presents several alternatives deemed helpful for the reader to appreciate differences, pros and cons.

2

The chapter is divided into three main loosely coupled sections, which may be read independently. Section 2.1 introduces some key concepts from the theory of learning. Section 2.2 describes some of the most common types of automata, mainly focusing on deterministic automata. Finally, Section 2.3 presents some state-of-the-art algorithms capable of learning automata from data.

2.1. THE PROBLEM OF LEARNING

The theory of learning, also known as learning theory or inductive inference, studies the problem of learning a concept from examples. This theory has two main actors: the student and the teacher. The student needs to learn and try to identify the concept from the examples. The teacher knows sufficient information about the concept allowing it to provide examples to the student by drawing them from the instance space (the input space). Based on the examples it has access to, the student formulates a hypothesis for the concept. The concept and the hypothesis are considered subsets of instances from the input space. The student is said to learn the concept when the teacher confirms that the hypothesis and the concept are the same subsets of the instance space.

In learning theory, it is expected to require the student to infer a hypothesis of minimal size. A smaller hypothesis is a simpler hypothesis, and among the possible concurrent hypotheses for the same concept, the simplest one is usually preferable because it is more interpretable. The minimal size requirement is just another application of Occam's razor principle, which is well-known in science. In learning theory, Occam's razor principle is adopted as a heuristic guiding the concept identification.

After describing the learning problem in general terms, it is worth defining those notions since the rest of the section builds up from them. The concept and hypothesis can be defined as follows:

Definition 1. A concept C over an instance space X is a subset of X. An example x is a positive example of C if $x \in C$. Otherwise, an example x is a negative example of C if $x \notin C$.

Definition 2. A hypothesis H for a target concept C over an instance space X is a subset of X. An example x evaluates true in H if $x \in H$; otherwise, it evaluates false. A hypothesis H for a target concept C is correct if and only if $H = C$.

Teacher and student are defined as follows:

Definition 3. A teacher t of a target concept C is an oracle that returns a labelled exam-

[1] Most of this chapter is based on the books by Verwer [81] and de la Higuera [61].

ple (x, b) where $x \in X$ is an example and b is a boolean value evaluated true if $x \in C$ and false otherwise.

Definition 4. A student s is a learning algorithm which learns a hypothesis H and has access to a teacher t of a target concept C. A student's goal is to find a correct hypothesis, i.e., a hypothesis H such that $x \in H$ if and only if $x \in C$.

The student has access to examples of the target concept without knowing it. However, it is commonly assumed that the student knows the class of the concept. In other words, the student knows the target concept has been chosen from a known concept class. A concept class is formally defined as follows:

Definition 5. A concept class $\mathscr{C}$ over an instance space X is a recursively enumerable set of concepts over X.

This section ends by introducing the notion of representation, defined as follows:

Definition 6. A representation for a concept class $\mathscr{C}$ is a function $\mathscr{R} : \mathbb{N} \to \mathscr{C}$. Any $i \in \mathbb{N}$ such that $\mathscr{R}(i) = C$ is called a representation of C under $\mathscr{R}$.

Representations heavily affect the complexity of a learning problem. Indeed, it is known that a learning problem which is computationally intractable using a specific representation may become treatable by choosing a different representation. This aspect is further discussed in Section 2.1.2.

2.1.1. LEARNING FRAMEWORKS

In learning theory, any learning process may adhere to three main learning frameworks. Each framework guarantees specific properties and requires the fulfilment of certain requirements.
In the identification in the limit framework, [82], a student learns a correct hypothesis from a teacher, eventually providing every sample. In the query learning framework [83], a student learns a correct hypothesis from a teacher by asking specific questions and reacting to the corresponding answers. In PAC identification framework [84], a student learns a probably approximately correct (PAC) hypothesis from a teacher who provides samples from an unknown probability distribution.
The choice of which learning framework to adopt depends on the application and the type of information at one's disposal. For example, finding a teacher capable of answering the questions required by the query learning framework can be challenging. A possible application is when a domain expert is available, and he/she is unable to codify his/her knowledge. PAC identification framework does not require such a type of teacher, but it imposes some substantial requirements on the student. Those requirements may make developing a PAC-compliant student very difficult in some cases. However, PAC identification guarantees that the student identifies a correct hypothesis with high probability. In most cases, accessing some samples of the target concept is relatively

easy. The identification in the limit can be applied without the substantial requirements imposed by the PAC identification framework. That is why most of the algorithms discussed later in this chapter work in the identification in the limit framework.
The following sections cover the three prominent learning frameworks in more detail.

IDENTIFICATION IN THE LIMIT

The identification in the limit [82] is very close to how humans learn since a human learner usually starts from a part of a concept. Eventually, the human learner extends his understanding by getting more information and learning more details. Identification in the limit models the concept of learning as a continuously ongoing process in which the student receives samples from the teacher, and it may change its mind. Intuitively, a mind change in the student happens when an instance represents a counterexample for the currently formulated hypothesis.
In finite identification, students can change their minds a limited number of times. The identification in the limit relaxes this requirement by allowing the students to change their minds infinitely often. However, when the student is allowed to learn with an infinite number of mind changes, it must converge to a hypothesis for the target concept. Convergence is a crucial property of any student in the identification in the limit framework because it guarantees that it is possible to understand whether the student has learned the target concept.f

Definition 7. A student identifies a concept C in the limit from a teacher t when it outputs a sequence of hypotheses $H_1, H_2, \ldots$ and there exists a number $n \in \mathbb{N}$ such that from that point onwards the output hypothesis remains the same, i.e. $H_n = H_{n+1} = H_{n+2} \ldots$ and H_n is a correct hypothesis.

A typical complexity measure for students who learn in the limit is the number of mind changes required to learn the target concept. In addition, the run-time of the student is also a common complexity measure. When a student s is guaranteed to identify any correct hypothesis H from a concept class $\mathscr{C}$ using polynomial run-time and amount of mind changes in the size of H, it is said that s efficiently identifies $\mathscr{C}$ in the limit.

QUERY LEARNING

Query learning framework [83] is characterized by a student asking questions to a teacher. Therefore, the first important aspect worthy of consideration is that query learning requires a teacher capable of answering those questions. Query learning is considered a type of active learning as opposed to all the frameworks which do not require queries. The latter are examples of passive learning.
A teacher in the query learning framework may answer four types of queries: membership queries, equivalence queries, subset queries, and superset queries. Membership queries take an element $x \in X$ as input, where X is the instance space, and return true if $x \in C$ where C is the target concept. Otherwise, if $x \notin C$, the teacher returns false. Equivalence queries take a hypothesis $H \subset X$ as input, and the output is yes if H is correct. Otherwise, if H is incorrect, the output is a pair (x, b) where $x \in (H \setminus C) \cup (C \setminus H)$ and b is a boolean value indicating if $x \in C$. The pair (x, b) is a counterexample certifying the

incorrectness of H. Subset queries take a hypothesis $H \subset X$ as input, and the output is yes if $H \subset C$. Otherwise, the output is a sample x where $x \in H \setminus C$. As before, x represents a counterexample certifying the incorrectness of H. Superset queries take a hypothesis $H \subset X$ as input, and the output is yes if $C \subset H$. Otherwise, the output is a sample x where $x \in C \setminus H$. Again, x represents a counterexample certifying the incorrectness of H. A teacher capable of answering membership and equivalence queries is called minimally adequate teacher.
The complexity of a query-learning student is commonly measured by the number of queries required to identify a concept. Of course, the run-time complexity of the student is also important when determining the complexity of a learning problem. When a student correctly identifies a hypothesis H from a concept class $\mathscr{C}$ with a polynomial number of queries in the size of H and with polynomial run-time in the size of H, it is said to identify $\mathscr{C}$ efficiently from queries.

PAC LEARNING

The main characteristic of the Probably Approximately Correct (PAC) learning framework [84] is the definition of a probabilistic teacher, which chooses samples from the input space according to an arbitrary distribution. The student needs to learn the sampling distribution. Another peculiarity of PAC learners is in the goal of the learning itself. While in the other frameworks, the goal is to identify the target concept exactly, PAC learning suffices to classify any new example with high accuracy.
As already mentioned in the introduction of the current Section, the PAC learning framework poses more requirements for the student than the other settings. A PAC-compliant student is a probabilistic student accepting three parameters as input: a probabilistic teacher drawing samples with an unknown probability distribution, a value ϵ denoting the maximum allowed probability error in classifying new samples, and a value δ denoting the minimum allowed probability of failure in identifying a hypothesis. Indeed, a PAC-compliant student must identify with probability at least $1-\delta$ a hypothesis H having probability at most ϵ of failing in classifying new samples.
The complexity of a PAC student is determined according to two parameters: a value k denoting an upper bound on the size of the hypothesis minimal representation and a value n denoting the size of the instance space. A student efficiently identifies a concept in the PAC learning framework if it runs in time polynomial w.r.t. k, n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$.

2.1.2. TIME COMPLEXITY

An essential question in learning theory is how complex learning problems are, regardless of the learning framework used to learn them. Although the actual complexity of a learning problem highly depends on the specificity of the problem itself, the process of attributing it to a specific complexity class is the same as for any other computational problem. In other words, the complexity of a learning problem is demonstrated by connecting it to another problem for which the complexity class is known. This process is called reduction.
In a reduction, any instance of problem A is transformed into an instance of a different problem B. If this transformation is efficient, i.e. polynomial in the size of any instance of A, and if the complexity class of B is known, then A has the same complexity class of B.

More intuitively, if A can be reduced to problem B, and if B can be solved efficiently, so is A because the algorithm for solving B could be used to solve A with the same efficiency. What makes a learning problem reducible to, in principle, any other computational problem is the notion of consistency formally defined as follows:

Definition 8. A concept $C \in \mathscr{C}$ is consistent with a finite set of labeled samples

$$S = \{(x_1, b_1), (x_2, b_2) \dots (x_n, b_n)\}$$

when for all $1 \leq i \leq n$, b_i = true if $x_i \in C$ and b_i = false otherwise.

With consistency, any instance of problem A can be mapped to a set of labelled samples S, which is then used to learn a class $C \in \mathscr{C}$. A condition for the reduction to properly work is that the size of S must be bounded by a polynomial of the size of the instance of A. In this way, any problem of learning a specific hypothesis can be interpreted as a search problem where the search space is composed of all possible hypotheses, and the goal is to find a hypothesis for which the error is zero.
As an example, it has been proved in [85] that the problem of learning three terms Disjunctive Normal Form (3DNF) formulas is NP-hard by reducing it to the problem of colouring graph nodes with three colours (also known as graph three colouring) which is also NP-hard. Therefore, finding an efficient algorithm for solving 3DNF formulas is only possible if $P = NP$.
Even when the target concept class for a learning problem is given, the choice of the hypothesis representation significantly impacts the efficiency of the learning algorithm. In other words, how a hypothesis is represented heavily affects the complexity of the learning problem. For example, the problem of learning three terms Conjunctive Normal Form (3CNF) formulas is tractable in the PAC learning framework. Therefore, by representing 3DNF formulas in conjunctive normal form, namely by using basic boolean algebra distribution rules, the problem of learning 3DNF formulas can be efficiently solved using a 3CNF hypothesis representation.

2.1.3. DATA COMPLEXITY

Like the time complexity, another essential question in learning theory is how much data is required to learn something meaningful. This question is also known as the data complexity of learning problems. The data complexity measures how many samples a student requires to learn a sensible hypothesis. A hypothesis is sensible if it is correct for at least some samples in addition to those provided by the teacher. In other words, a sensible hypothesis should prove that the student learned at least something.
Given a set of samples S, also called a training set, a labelling S_l of S is a set of all elements from S together with a boolean value. In other words, S_l is a labeling of S if and only if $S = \{x \mid \exists b \in \{\text{true}, \text{false}\} \text{ s.t. } (x, b) \in S_l\}$.

A concept class $\mathscr{C}$ achieves a labelling S_l if there exists a concept $C \in \mathscr{C}$ such that $x \in C$ if and only if $(x, \text{true}) \in S_l$. When a concept class $\mathscr{C}$ achieves all possible labellings of a training set S, S is shattered by $\mathscr{C}$. In other words, if a concept class $\mathscr{C}$ is shattered by a training set containing n samples, a student requires at least n samples to learn a sen-

sible hypothesis. When a hypothesis H is learned from a training set S' with cardinality $n-1$, for every other sample $x \notin S'$ there exists a hypothesis H' consistent with $S' \cup \{x\}$ and a hypothesis H' not consistent with $S' \cup \{x\}$.
The Vapnik-Chernovenkis dimension (VCD) of a concept class $\mathscr{C}$ is the cardinality of the largest training set shattered by $\mathscr{C}$ [86]. The VCD is interesting because it is used to derive a bound on the number of samples required to learn a certain concept in the PAC learning framework. Indeed, any hypothesis $h \in \mathscr{H}$ consistent with a sample of size

$$\max\left(\frac{4}{\epsilon}\log\left(\frac{1}{\delta}\right), \frac{8}{\epsilon}VCD(\mathscr{H})\log\left(\frac{13}{\epsilon}\right)\right)$$

has error at most ϵ with probability at least $1-\delta$. However, that bound overestimates the actual bound for most learning problems. Commonly, a much smaller training set suffices to converge to a sensible hypothesis for a concept class.

2.2. AUTOMATA MODELS

This section focuses on automata models, and there are at least two good reasons of interest in them[2]. The first reason is that many real-world systems, both hardware [87, 88] and software [56, 76], can be considered Discrete Event Systems (DESs), and automata are the tool of choice when modelling DESs. A DES is a system of states associated with a finite set of events. A DES has three fundamental properties: it is in a single state at any time during its life cycle; an event may occur instantly, and it always causes a transition from one state to another (possibly the same).
The second reason is about the learnability, i.e. the capability of learning those models from data. Several models are more expressive than automata, e.g. Turing Machines, but they are challenging to learn. Although the automata's expressive power is limited to the class of regular languages, they belong to the set of models that are provably and efficiently learnable from data.
This section defines and describes three different classes of automata, depending on the type of queries they aim to answer. Finite State Automata (FAs), treated in Section 2.2.1, are designed to answer the question, "Is a string belonging to a given regular language?". Instead, Probabilistic Automata (PAs) are designed to answer the question, "What is the probability of a string belonging to a given regular language?". Section 2.2.2 introduces the PAs. Finally, Timed Automata (TAs), addressed in Section 2.2.3, answer the same question of the FAs but for timed regular languages.

2.2.1. FINITE STATE AUTOMATA

Finite State Automata (FAs) are computational models requiring a finite amount of memory. A FA is composed of states and transitions. FAs can be graphically described as directed graphs with states represented by circles and transitions represented by arrows linking source states to destination states. Figure 1.4 shows an example of FA. FAs operate on sequences of symbols drawn from a set called alphabet. Each transition is labelled with a symbol. A computation is the process of changing the current state of an FA by executing its transitions. Therefore, an FA computation can be represented as a sequence

[2] In addition to the obvious consideration that automata are one of the core topics of this thesis.

of states and transitions activated during its life cycle.

Initially, the current state of an FA is the so-called initial or start state (graphically represented by a state pointed by a sourceless transition). Unless a state is not reachable from the initial state, each transition in the computation is activated, and that causes a change in the current state to the destination state pointed by that transition. The destination state is also called the next state because it follows the source state in the computation. A computation on an FA is said to be valid if and only if it ends in a final state. The final states are a subset of an FA's states and are graphically highlighted with double circles. All the labels of the transitions, activated in a computation, form a sequence of symbols called string.

There are three different types of FAs depending on the computation semantics. In an Acceptor, any computation accepts or rejects a string, which is achieved by defining two additional sets of states: accepting and rejecting. If a computation ends in an accepting state, the string, composed of the labels of the transitions activated by the computation, is accepted. Otherwise, the string is rejected if the computation ends in a rejecting state. Generators can be considered a type of Acceptor where the alphabet is composed of just one symbol. Generators can only emit a single string since, at most, one transition is leaving each state. In a Transducer, any computation generates an output string starting from an input string. Transducers do not define accepting or rejecting states. Instead, they define two distinct alphabets (i.e., the input and output alphabet), and the transitions are decorated with an additional symbol taken from the output alphabet. Any activated transition maps an input symbol to the corresponding output symbol during a computation. In this way, computations map input strings to output strings.

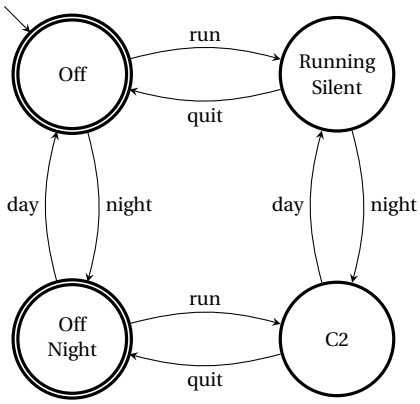


Figure 1.4: An automaton representing an over-simplified malware behaviour. It connects to the Command & Control server (C2) if and only if it gets executed and it's nighttime (originally from page 13).

The set of strings a FA $\mathscr{A}$ accepts[3] is called the language of the FA, and it is denoted

[3] Any consideration expressed in this section holds for either Acceptors, Generators or Transducers. Therefore, any reference to string acceptance or language accepted is replaceable with, respectively, string generation

by $L(\mathscr{A})$. A FA accepts a string if it is formed by the labels of the transitions activated on a valid computation. In this case, the string is said to be valid as well. Therefore, the language accepted by an FA is the set of valid strings according to the FA itself. It is important to distinguish between deterministic finite state automata (DFAs) and non-deterministic finite state automata (NFAs). In DFAs, each state has at most one transition activable with the same label. Therefore, there are no ambiguities in a DFA on which transition to activate in a computation. In NFAs, this constraint is relaxed, and multiple transitions leaving a state can be labelled with the same symbol. Therefore, there can be multiple options in an NFA at any point in a computation. It is worth considering that this difference does not affect which languages may or may not be accepted by FAs. Instead, (non-)determinism influences language representation. It is also important to distinguish between FAs accepting finite-length strings and FAs accepting infinite strings. The most common type of automaton accepting infinite strings is the Büchi automaton.

DETERMINISTIC AUTOMATA

DFAs are the most common type of automata. DFAs can be considered a building block for other models, such as Probabilistic Automata or Timed Automata, which share a similar same structure. A DFA is formally defined as follows:

Definition 9. A DFA is a 5-tuple $\left(Q, \Sigma, q_0, \delta, F\right)$, where

- Q is a finite set of states.
- Σ is a finite set of symbols, aka the alphabet.
- $q_0 \in Q$ is the start state.
- $\delta : \Sigma \times Q \rightarrow Q$ is the transition function.
- $F \subseteq Q$ is a set of final states.

The transition function δ defines the transitions in a DFA. A DFA uses transitions to generate or accept strings depending on the type of the automaton. Strings are accepted or generated by computations. The notion of computation of DFAs is defined as follows:

Definition 10. A computation of a DFA $\mathscr{A} = \left(Q, \Sigma, q_0, \delta, F\right)$ over a string $a_1, a_2, \ldots a_n$ is a sequence

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots \xrightarrow{a_n} q_n$$

of states and transitions such that $\delta(a_i, q_{i-1}) = q_i$ for all $i = 1 \ldots n$, $q_i \in Q$ and $a_i \in \Sigma$. A DFA computation is valid if $q_n \in F$.

The transition function is partial since not all the symbols may activate a transition in a given state. Considering the DFA in Figure 1.4, the symbol day does not activate any

and language generation.

transition in state Off. Those types of symbols are infeasible instead of those that activate a transition (such as, for example, *run* in state Off of the automaton in Figure 1.4), which are feasible. A DFA generates a language if it comprises all the strings for which a feasible computation exists. A DFA recognizes a language if it is composed of all the strings for which there is a valid computation. The recognized language of a DFA is a subset of the generated language for the same DFA. Figure 1.4 shows an instance of DFA.

NON-DETERMINISTIC AUTOMATA

In an NFA, determining the current state may not be possible. That is the difference between a NFA and a DFA. Indeed, in an NFA, it is correct to talk about the set of next states instead of the next state as for DFAs. The first cause of non-determinism occurs when the current state may change to two or more possible next states given a symbol. A second cause of non-determinism occurs when the current state may change without any observed symbol occurrence. The latter case is graphically modelled with transitions labelled with the empty string symbol ϵ and because of that said ϵ-transitions. In more formal terms, a NFA is defined as follows:

Definition 11. A NFA is a 5-tuple $(Q, \Sigma, Q_0, \delta, F)$, where

- Q is a finite set of states.
- Σ is a finite set of symbols, aka the alphabet, united with the ϵ symbol to denote unobserved symbols.
- $Q_0 \subseteq Q$ is a set of possible start states.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is the transition function.
- $F \subseteq Q$ is a set of final states.

As it is possible to notice from the definition above, the initial state of the DFA definition has been replaced by a set of possible initial states in the NFA definition. In addition, the transition function in NFAs maps to sets of states instead of states as for DFAs. Both modifications result from the non-determinism of the state of the automaton. In an NFA, there can be multiple computations given the same string. That is why the formal definition of computation of NFAs changes as follows:

Definition 12. A computation of an NFA $\mathscr{A} = (Q, \Sigma, Q_0, \delta, F)$ over a string $a_1, a_2, \ldots a_n$ is a sequence

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ldots \xrightarrow{a_n} q_n$$

of states and transitions such that $q_0 \in Q_0$, $q_i \in \delta^*(\epsilon^* a_i, q_{i-1})$, $q_i \in Q$ and $a_i \in \Sigma \cup \{\epsilon\}$, for all $i = 1 \ldots n$. An NFA computation is valid if $q_n \in F$.

δ^* is defined as the extension of δ to string inputs. As such, δ^* is defined as the partial mapping of $\Sigma^* \times Q$ into 2^Q. Figure 2.1 shows an instance of NFA.

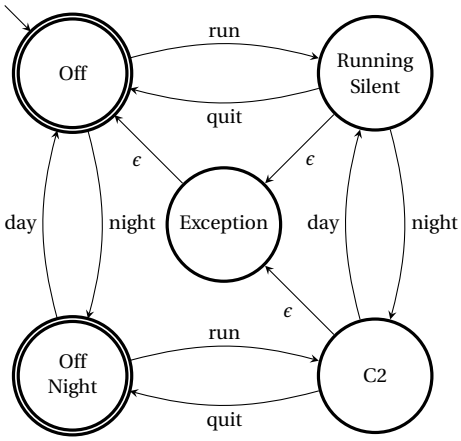


Figure 2.1: A non-deterministic finite state automaton representing a possible malware behaviour. After being installed, it connects to the Command & Control server (C2) if only if it gets executed and it is night time. Whenever in a running state, the malware may raise an exception, which causes the software to crash.

2.2.2. PROBABILISTIC AUTOMATA

In many real-world applications, knowing the probability of a string belonging to a language is more important than if it belongs to the language. Probabilistic Automata responds to this type of use case. To this extent, PAs are considered generative devices rather than parsing or recognizing devices as FAs. In this section, we define Probabilistic Non-deterministic Automata (PNAs). We introduce their deterministic counterpart, the Probabilistic Deterministic Automata (PDAs), in Chapter 7.

Definition 13. A PNA is a 5-tuple $(Q, \Sigma, I_P, F_P, \delta_P)$, where

- Q is a finite set of states.
- Σ is a finite set of symbols, aka the alphabet.
- $I_P : Q \to [0,1]$ is the initial state probability distribution.
- $F_P : Q \to [0,1]$ is the final state probability distribution.
- $\delta_P : Q \times Q \times (\Sigma \cup \{\epsilon\}) \to [0,1]$ is the transition function. This function is complete, i.e., $\delta(q, q', a) = 0$ with $q, q' \in Q$ and $a \in \Sigma$ can be interpreted as no transition existing from q to q' with symbol a.

The three probability distributions I_P, F_P, and δ_P, are functions such that

$$\sum_{q \in Q} I_P(q) = 1$$

and $\forall q \in Q$,

$$F_P(q) + \sum_{q' \in Q,\, a \in \Sigma \cup \{\epsilon\}} \delta_P(q, q', a) = 1.$$

I_P, F_P, and δ_P play a key role in defining the notion of computation in PNAs, which generate strings together with a probability of that string being generated. The formal definition of the PA-computation of PNA follows:

2

Definition 14. A computation of a PNA $\mathscr{A} = (Q, \Sigma, I_P, F_P, \delta_P)$ over a string $a_0, a_2, \ldots a_n$ is a pair (r, p) consisting of:

- A finite sequence of states and transitions $r = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots q_{n-1} \xrightarrow{a_{n-1}} q_n$.
- A probability value $p = I_P(q_0) \times \prod_{0 \leq i \leq n-1} \delta_P(q_i, q_{i+1}, a_i) \times F_P(q_n)$ where $q_i \in Q$ for all $0 \leq i \leq n$ and $a_i \in \Sigma \cup \{\epsilon\}$ for all $0 \leq i \leq n-1$.

By comparing the definition of PNAs with NFAs of the previous section, it is possible to notice that there are no accepting or rejecting states in NFAs. That holds for any PAs since they are generative in nature.

The following section discusses some widely used models that can be represented as PAs.

MARKOV CHAIN, NGRAM MODEL AND HIDDEN MARKOV MODELS

A few widely used models can be expressed as PAs. A Markov Chain [89] is a stochastic process $\{X_n | n = 0, 1, 2, \ldots\}$ admitting a finite or countable number of positive values. When $X_n = i$, the process is said to be in state i at time n. When a Markov Chain is in state i at time n there is a probability $P_{i,j}$ that it will transition into state j at time $n+1$. That is a fundamental characteristic known as Markov property: given the current state, the future state is independent of the past. That is why it is said that Markov Chains are memoryless processes.

Markov Chains may be represented as Probabilistic Automata when the number of states is finite. However, since Markov Chains only models the probability of being in a state after some time, only the states are labelled. Differently from what was shown in the previous section, PAs representing Markov Chains have un-labelled transitions.

The Ngrams model, or simply Ngrams, is a common type of Markov Chain used in language learning applications. An Ngrams model comprises states, each denoting a finite sequence of past symbols of length N. In Trigrams, an instance of Ngrams with $N = 3$, a state exists for each sequence of three alphabet symbols. Furthermore, Ngrams are composed of transition probabilities between states to express the probability of the next symbol. For example, in Trigrams from state abc to state bcd, there can be a non-zero probability $P(d|abc)$ of d being the next symbol given the current state abc.

Hidden Markov Model (HMM) [90] is essentially a Markov Chain where the states are unobservable. From this aspect comes the name of this type of model. Each state of an HMM has a probability of emitting a given observation. Each state has its emission probability distribution, independent of the transition probability. In HMMs, only the observations are visible from outside the model. HMMs may be described with PAs even if their transitions are unlabeled. Indeed, the probability distributions created by HMMs are identical to the ones created by PAs [91].

2.2.3. TIMED AUTOMATA

All the automata described in the previous sections neglect the temporal dimension, namely any temporal relationship among the alphabet symbols. However, time information is often a crucial aspect of real-world systems. Time relationships among symbols are modelled by adding a timestamp to each symbol occurrence. Sequences of these timestamped symbols are called timed strings, formally defined as follows:

Definition 15. A timed string τ over an alphabet Σ is a sequence of pairs

$$(a_1, t_1), (a_2, t_2) \dots (a_n, t_n)$$

where $a_i \in \Sigma$ and $t_i \in \mathbb{R}_+$, $i = 1 \dots n$, are the symbols and the time values, respectively. The untimed string corresponding to a timed string $(a_1, t_1), (a_2, t_2) \dots (a_n, t_n)$ is the string $a_1, a_2 \dots a_n$ obtained by removing the time values.

The time values $t_1, t_2, \dots, t_n$ are obtained by evaluating one or more clocks. A clock is an object with the following properties:

- It increases over time, synchronously with other clocks.
- It is resetable to 0. After being reset, a clock will immediately start increasing. We denote the operation of clock resetting as reset.
- It can be evaluated. That means there is a function that maps the clock to its current value.

The clock evaluation function is commonly denoted as v with $v(x) \in \mathbb{R}_+$ being the evaluation for clock x. A clock guard, defined as follows, can express time constraints.

Definition 16. A clock guard is an arithmetic constraint on clocks inductively defined as

$$\begin{array}{rcl} g & := & x \leq n \\ & | & n \leq x \\ & | & x \leq y \\ & | & g_1 \vee g_2 \\ & | & g_1 \wedge g_2 \end{array}$$

Where x and y are clocks, $n \in \mathbb{Q}$, and g_1 and g_2 are clock guards. A clock guard is satisfied by a clock valutation when the guard evaluates to true given the clock values.

For example, the clock guard $x \geq 30$ for the clock x represents the time constraint being satisfied if the value of the clock x is greater than 30 seconds. Therefore, the clock evaluation $x = 3$ does not satisfy the time guard, and the evaluation $x = 45$ does.

A Timed Automaton (TA), in its original definition introduced in [92], is a class of automata with no constraint on the number of clocks. Furthermore, each transition of a TA is capable of resetting any number of clocks. A TA is defined as follows:

Definition 17. A TA is a 6-tuple $\left(Q, X, \Sigma, \Delta, q_0, F\right)$, where

- Q is a finite set of states.
- X is a finite set of clocks.
- Σ is a finite set of symbols, aka the alphabet.
- Δ is a finite set of transitions.
- q_0 is the start state.
- $F \subseteq Q$ is a set of final states.

A transition $\delta \in \Delta$ is a 5-tuple (q, q', a, g, R), where

- $q \in Q$ is the source state.
- $q' \in Q$ is the destination state.
- $a \in \Sigma$ is a transition label symbol.
- g is a clock guard.
- $R \subseteq X$ is the set of clock resets.

As it is possible to notice from the definition above, each transition δ in a TA includes a set of clocks R. When a timed symbol activates δ, the values for all the clocks in R are set to 0, therefore $\forall x \in R$, $v(x) := 0$), while the other clocks' values are unaltered. This dynamic allows for recording how much time elapsed since the occurrence of a specific symbol. The behaviour of a TA is conditioned by the clock values as formalized in the following definition of TA-computation:

Definition 18. A computation of a TA $\mathscr{A} = (Q, X, \Sigma, \Delta, q_0, F)$ over a timed string

$$\tau = (a_1, t_1), (a_2, t_2) \dots (a_n, t_n)$$

is a finite sequence of states and transitions

$$q_0 \xrightarrow{(a_1,t_1)} q_1 \xrightarrow{(a_2,t_2)} q_2 \dots q_{n-1} \xrightarrow{(a_n,t_n)} q_n$$

such that there exists a transition $(q_{i-1}, q_i, a_i, g, R_i) \in \Delta$ with g satisfied by the evaluation v_i for all $i = 0, 1 \dots n$, $q_i \in Q$, and $a_i \in \Sigma$. The evaluation v_i is defined as $v_i(x) := 0$ if $x \in R_i$ else $v_i(x) := v_{i-1}(x) + t_i$ and $v_0(x) := 0$ for all $x \in X$. A finite computation of a TA is valid when $q_n \in F$.

Real-Timed Automata (RTAs) [93] are a particular type of timed automata that recur to just one clock because they model the time delay between two consecutive symbols. Clock guards in RTAs represent time delays in terms of intervals and are also called delay guards. A formal definition of RTA follows:

Definition 19. An RTA is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$ where

- Q is a finite set of states.
- Σ is a finite set of symbols, aka the alphabet.
- Δ is a finite set of transitions.
- q_0 is the start state.
- $F \subseteq Q$ is a set of final states.

A transition $\delta \in \Delta$ is a 4-tuple $(q, q', a, [n, n'])$ where

- $q \in Q$ is the source state.
- $q' \in Q$ is the destination state.
- $a \in \Sigma$ is a transition label symbol.
- $[n, n']$ is a delay guard with $n, n' \in \mathbb{N}$.

There is no need to explicate the single clock in the RTA definition since it is unnecessary to reset and evaluate it. A transition $(q, q', a, [n, n'])$ in an RTA is interpreted as follows: whenever the automaton's current state is q, after reading a timed symbol (a, t) such that $a \in \Sigma$ and $t \in [n, n']$, the automaton state changes to q'. An RTA is said to be deterministic if it does not contain two transitions with the same label, the same source state, and overlapping delay guards. The behaviour of an RTA is influenced by the delay guards, as shown in the following definition of RTA-computation.

Definition 20. A computation of an RTA $\mathscr{A} = \big(Q, \Sigma, \Delta, q_0, F\big)$ over a timed string

$$\tau = (a_1, t_1), (a_2, t_2) \dots (a_n, t_n)$$

is a finite sequence of states and transitions

$$q_0 \xrightarrow{(a_1,t_1)} q_1 \xrightarrow{(a_2,t_2)} q_2 \dots q_{n-1} \xrightarrow{(a_n,t_n)} q_n$$

such that $(q_{i-1}, q_i, a_i, [n_i, n_i']) \in \Delta$, where $t_i \in [n_i, n_i']$, for all $i = 1, 2 \dots n$. An RTA computation is valid when $q_n \in F$.

An RTA $\mathscr{A}$ accepts a timed string τ if the computation of $\mathscr{A}$ over τ ends in an accepting state q_n, i.e. $q_n \in F$. The language of an RTA $\mathscr{A}$, denoted as $L(\mathscr{A})$, is the set of timed strings for which there exists an accepting computation. Figure 2.2 shows an instance of an RTA.

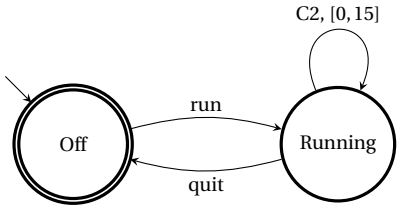


Figure 2.2: A Real-Time Automaton representing a possible malware behaviour. After being launched, the malware tries to connect to the Command & Control server (C2) once every at least 15 minutes. When omitted, the transition guards are intended as unbounded, i.e., ranging from the minimal to the maximal delay possible.

2.3. LEARNING AUTOMATA FROM DATA

This Section proposes several widely known algorithms for learning automata from data. The problem of learning a model from data is known as system identification. Here, the target model is an automaton. Therefore, the problem is also called automata identification. Furthermore, the problem of automata learning is also known as grammatical inference. That is why the words inference and identification throughout the dissertation are used as synonyms for learning unless differently and explicitly stated.

The first essential premise worth discussing is that the entire Section is devoted to algorithms assuming the target model is unavailable. Target model unavailability is a typical setting known as passive learning, as opposed to the active learning setting, which is about learning from querying a running system. Learning automata from network data, one of this work's critical contributions, is a passive learning setting, which is why this Section focuses on these algorithms.

A second but not less critical premise is about the interest in learning the structure of the model, i.e. how many states the model has and how they are transition-linked. In Machine Learning and Artificial Intelligence, people often learn the parameters of their models and assume the structure. For example, in Neural Networks, the structure of the model is pre-defined by the domain experts and the learning algorithm is limited to the network parameters inference - e.g. weights estimation. In those cases, the learning problem is relatively simple. However, besides the problem's complexity, learning the structure is deemed necessary whenever the system is unknown, and there is a need to gain insight into it.

All the algorithms discussed in the following sections are for learning deterministic models. In the implementations of the proposed methodology explored in the following chapters, we decided to focus on deterministic models because of the availability of well-performing algorithms for learning them. The motivation for such a choice is all about the complexity of the learning problem. Indeed, even if automata are learnable from data, the problem of learning them is still challenging. If learning DFAs can be an NP-Hard problem, then learning NFA is even more complex.

Section 2.3.1 discusses the most common algorithm for learning DFAs. This algorithm is important because it has been specialized to learn other types of automata, such as Probabilistic Automata and Timed Automata. Section 2.3.2 discusses a common algorithm to learn Probabilistic Deterministic automata. Finally, Section 2.3.3 discusses

the state-of-the-art algorithm for learning Probabilistic Deterministic Real-Timed Automata.

2.3.1. STATE MERGING

An automaton learning algorithm aims to find an automaton $\mathscr{A}$, also called the hypothesis, such that its language $L(\mathscr{A})$ is equal to the language of the automaton target of the learning process. This latter automaton is simply referred to as the target, and its language is denoted as L_t. Therefore, the goal of an automaton learning algorithm is to find $\mathscr{A}$ such that $L(\mathscr{A}) = L_t$. A common assumption is that a set of both positive and negative samples is given. A positive sample is a string s such that $s \in L_t$, and a negative sample is a string s such that $s \notin L_t$. The positive sample is denoted as S_+, and the negative sample is denoted as S_-. The set $S = \{S_+, S_-\}$ is called the training set.
This section discusses the State Merging algorithm, which aims to find the smallest DFA possible consistent with the training set, i.e. accepting all the positive samples in S_+ and rejecting all the negative samples in S_-. It seeks the smallest DFA because of Occam's razor principle. A more straightforward explanation is preferable because it makes the least number of assumptions. In the world of automata, simplicity maps to the number of states, i.e. the most straightforward hypothesis for a target is also the smallest. The search for the smallest DFA consistent with the training set produces models which are easier to understand.

Algorithm 1 Construct the APTA: apta

Require: A training set $S = \{S_+, S_-\}$
Ensure: An APTA $\mathscr{A}$ for S
  Initialize $\mathscr{A}$ with the only start state q_0
  $A := \emptyset$ ▷ A is the set of accepting states in $\mathscr{A}$
  $R := \emptyset$ ▷ R is the set of rejecting states in $\mathscr{A}$
  for each sample $s = a_1, a_2 \dots a_n$ from S **do**
    $q' := q_0$, $i := 0$
    while $i \leq n$ **do**
      if $\nexists q \in Q$ s.t. $\delta(q', a_i) \rightarrow q$ **then**
        $Q := Q \cup \{q''\}$ ▷ Create a new state q''
        $\delta(q', a_i) := q''$
      $q' := \delta(q', a_i)$, $i := i + 1$
    if $s \in S_+$ **then** $A := A \cup \{q'\}$ **else** $R := R \cup \{q'\}$
  $F := A \cup R$
  return $\mathscr{A}$

The idea of the State Merging algorithm is to start by constructing a first hypothesis for the target. This DFA is called Augmented Prefix Tree Acceptor (APTA), and the Listing 1 shows the construction procedure. An APTA is a prefix tree because it contains one path from the start state to any other state and because the computation of two strings s and s' reach the same state if and only if they share the same prefix. Furthermore, an APTA is a DFA consistent with the training set S, called augmented, because it usually contains

states that are neither accepting nor rejecting. It is impossible by construction that an execution of any sample from S ends in such undecided states, and the algorithm tries to resolve those undecisions by merging states.
The merging operation combines two states q and q' into one by creating a new state q'' that has the incoming and outgoing transitions of both q and q'. By merging states, the size of the DFA decreases, allowing the algorithm to approach the smallest DFA given the training set. The merging operation is performed if and only if q and q' are considered consistent. Two states are consistent if it is not the case that one is an accepting state and the other is a rejecting state. Merging in case of inconsistency would violate what was observed in the training data. This violation could lead to accepting a string known as not belonging to L_t or vice-versa to reject a string known as belonging to L_t. In other words, merging inconsistent states leads to the formation of a wrong hypothesis.
The merging operation may introduce a non-determinism. That happens when a merge is accomplished between two states, both having an outgoing transition labelled with the same symbol and reaching different destination states. In such cases, a so-called determinization operation must be taken to satisfy the deterministic property. A typical determinization operation consists of merging the destination states in a chain of merges that only ends when there are no non-deterministic choices left. The merging procedure is shown in Listing 2, with the final while loop implementing the determinization procedure.

Algorithm 2 Merging two states: merge

Require: An augmented DFA $\mathscr{A}$ and two states $q, q' \in Q$
Ensure: If q and q' are consistent then q and q' are merged, $\mathscr{A}$ is updated, and true is returned. False is returned otherwise.

```
if q ∈ A and q' ∈ R or vice-versa then return false
Q := Q ∪ {q''}                                   ▷ Create a new state q''
if q ∈ A or q' ∈ A then A = A ∪ {q''}            ▷ A is the set of accepting states in 𝒜
if q ∈ R or q' ∈ R then R = R ∪ {q''}            ▷ R is the set of rejecting states in 𝒜
for each δ(q_s, a) = q_d s.t. q_s ∈ {q, q'} or q_d ∈ {q, q'} do
    if q_s = q or q_s = q' then q_s := q''
    if q_d = q or q_d = q' then q_d := q''
while ∃δ(q_s, a) = q_n, δ(q_s, a) = q'_n do      ▷ While exists a non-deterministic choice
    e := merge(𝒜, q_n, q'_n)
    if e is false then
        Undo the merge between q and q'
        return false
return true
```

A State Merging algorithm continually applies the merging operator until no more consistent merges are possible. The red-blue framework for state merging follows the just-described principles. On top of what is already discussed, the red-blue framework is based on an abstraction of the hypothesis. This abstraction consists of a set of red states, also called the red core, surrounded by a fringe of blue states. The red core is the part of the hypothesis that has already been identified and is, therefore, finalized. The blue

fringe represents the part of the hypothesis composed of states discovered but not finalized yet. Only merges between red and blue states are allowed in the red-blue framework. Listing 3 shows how a State Merging algorithm in the red-blue framework works.

Algorithm 3 State Merging in the red-blue framework

Require: A training set $S = \{S_+, S_-\}$
Ensure: $\mathscr{A}$ is a small DFA consistent with S
  $\mathscr{A}$:= apta(S)
  $Red := Red \cup \{q_0\}$ ▷ Red is the red core of $\mathscr{A}$
  for each $q \in Q$ s.t. $\exists \delta(q_0, a) = q$ **do**
    $Blue := Blue \cup \{q\}$ ▷ $Blue$ is the blue fringe of $\mathscr{A}$
  while $Blue \neq \emptyset$ **do**
    Choose a blue state q_b
    if $\exists q_r \in Red$ s.t. merge($\mathscr{A}, q_r, q_b$) = true **then**
      Commit the merge between q_r and q_b
    else
      $Blue := Blue \setminus \{q_b\}$, $Red := Red \cup \{q_b\}$
  return $\mathscr{A}$

The state-of-the-art State Merging algorithm is the Evidence Driven State Merging (EDSM) algorithm [94]. EDSM adopts the red-blue framework, and its principal characteristic is that an evidence value guides the decision on which red-blue states to prioritize for merging. The evidence value quantifies the support for a candidate merge coming from the training set. The principle is to prioritize the merges which have passed the most tests with the hope that they are more likely to be correct. Indeed, the EDSM score for a candidate merge equals the number of merges it causes between accepting states and accepting states and the number of merges it causes between rejecting states and rejecting states. EDSM algorithm tries to avoid bad merges, i.e. merges with a weaker support.

2.3.2. ALERGIA

Learning Probabilistic Deterministic automata (PDAs) is a different problem from learning DFAs. In DFA learning, it is required to have positive and negative samples, i.e. $S = \{S_+, S_-\}$, to infer the target model. Having just positive samples makes it impossible to exclude that the target language does not consist of all the strings. Indeed, it has been proved by Gold [82] that identifying a DFA in the limit from positive samples (i.e. $S = \{S_+\}$) is impossible.
However, since in PDA learning, the ultimate goal consists of inferring a probability distribution over strings, there is no real need for negative samples, and the target model can be identified just from positive samples. Indeed, algorithms can learn PDAs in the limit with probability one. One of those algorithms is ALERGIA [95], which is the topic of this section.
Assume a training set S composed of positive samples only and assume the size of S be-

ing n, i.e. $|S| = n$. Furthermore, suppose to have observed a particular string s occurring t times into S. The goal of any PDA learning algorithm, including ALERGIA, is to build a model in such a way that it will assign a probability $\frac{t}{n}$ to the string s. That should hold for each string in S. In other words, the problem of identifying a PDA consists of finding the PDA that could have generated the distribution found in the training set. Furthermore, as for DFAs, it is required that the identified model should be the smallest possible.
The latter constraint is required for several reasons. For example, larger PDAs can produce more possible distributions, which may affect their compliance with the observations. Another reason is that smaller PDAs represent more straightforward explanations for the same phenomenon and, therefore, are preferable per Occam's razor principle. In practical terms, requiring the PDA to be the smallest possible improves model understanding.

ALERGIA is based on the State Merging algorithm for DFAs described in Section 2.3.1 as most of the algorithms for learning PDAs. Similarly to what happens in State Merging, ALERGIA starts with constructing a so-called Probabilistic Prefix Tree Acceptor (PPTA). A PPTA is a prefix tree representing the distribution of the training set S. Indeed, the PPTA construction procedure is almost identical to the APTA construction procedure shown in Listing 1. The only difference is the frequency counts collected by the PPTA procedure, which are required to estimate the probabilities. The estimator implemented in ALERGIA is a maximum likelihood estimator because it sets the probability of string $s = a_1, a_1 \dots a_n$ as follows:

$$P(s) = \prod_{1 \le i \le n} \frac{\text{count}(q, a_i)}{\text{count(q)}}$$

Where count(q, a), for a symbol a and a state q, represents the number of samples activating the transition $\delta(q, a)$ and count(q) represents the number of samples leaving the state q. The PPTA construction procedure keeps those counts up to date while processing all the samples in S.
Also, the merging procedure is the same as in State Merging, with the only addition of the frequencies updating after a merge between two states. ALERGIA's principal difference lies in the compatibility check. Since there are no negative samples in S, the consistency check implemented in State Merging is no longer applicable because there is no notion of rejecting state. Therefore, in ALERGIA, two states are deemed compatible for merging if they satisfy the Hoeffding bound statistical test.
Two states q and q' are compatible in ALERGIA, with confidence $\alpha > 0$, if the following condition holds for all $a \in \Sigma$, for both states and for any of their children:

$$\left| \frac{\text{count}(q, a)}{\text{count}(q)} - \frac{\text{count}(q', a)}{\text{count}(q')} \right| < \sqrt{\frac{1}{2} \ln\left(\frac{2}{\alpha}\right)} \left(\frac{1}{\sqrt{\text{count}(q)}} + \frac{1}{\sqrt{\text{count}(q')}} \right)$$

Another difference between ALERGIA and State Merging is the stopping criterion, i.e. when the algorithm stops and returns its hypothesis of the target PDA. The latter stops whenever all possible merges are inconsistent. The former stops whenever all possible merges are un-compatible as for the above-reported test.

2.3.3. RTI

As discussed in Section 2.2.3, Deterministic Real-Time automata (DRTAs) can model time delays between consecutive symbols. They achieve this goal by imposing a temporal constraint on each transition. Those constraints assume the form of delay guards controlling whether a transition is active, given a source state and a symbol. The problem of learning DRTAs poses a new challenge in comparison to the problem of learning Deterministic Finite states automata (DFAs). Indeed, when learning DRTAs, in addition to the problem of identifying the correct automata structure, there is a further problem of inferring the correct delay guards.
There could be the growing temptation to separate those two problems, solve them independently and eventually combine the corresponding solutions to solve the more general problem of identifying the DRTA. Although this divide-et-impera approach may seem promising, it is unfeasible. Indeed, it has been proved [81] that given a DRTA $\mathscr{A}$ and a training set S, the problem of inferring the correct delay guards such that $\mathscr{A}$ is consistent with S is NP-complete.
A more effective approach is to solve both problems by identifying the delay guards in a way similar to how an automaton's structure is determined. That is the strategy implemented by RTI [81], the state-of-the-art algorithm for identifying DRTAs from training sets composed of both accepted and rejected timed strings. This Section presents RTI and discusses how it achieves its goal.
RTI is based on the State Merging algorithm for identifying DFAs presented in Section 2.2.1. As such, it implements the red-blue framework by following an evidence-driven approach as EDSM does for FSAs. The general idea behind RTI is that it learns the automaton structure using the same State Merging tools, namely the merging operator between states. However, it also learns the delay guards by splitting the transitions thanks to an additional splitting operator.

Similarly to the State Merging algorithm, RTI starts by building a Timed Augmented Prefix Tree Acceptor (TAPTA). A TAPTA is called augmented because it contains additional information about the rejecting states. It is a Timed-APTA because its transitions are controlled by delay guards expressing temporal constraints. An important problem with the TAPTA construction is setting the delay guards. Essentially, there are two approaches: bottom-up and top-down.
In a bottom-up approach, the delay guards are set to the smallest interval possible, i.e. just the single value observed in some timed string within the training set. The goal of the subsequent steps of the algorithm would be to enlarge those intervals by merging transitions according to the evidence values. This approach poses a few challenges. Setting the delay guards to cover just one value means that most of the time, there is a branch in the TAPTA for each timed string in the training set. That happens because it is very likely that two timed strings have different time parts, even when the symbolic part is the same. In this situation, the merging operator would rarely merge two states since that case would only happen when the same timed string reaches them, i.e. all the merges would fail at the determinization step. Consequently, the evidence value in an EDSM-based algorithm is either 0 or 1, conveying little to no information.
In a top-down approach, the delay guards are initially set to the most general interval possible. Eventually, in subsequent algorithm steps, they may get specialized by split-

Algorithm 4 Construct the timed APTA: tapta

Require: A training set $S = \{S_+, S_-\}$ with alphabet Σ and having minimum delay value t_{min} and maximum delay value t_{max}
Ensure: $\mathscr{A}$ is a timed APTA for S
Initialize $\mathscr{A}$ with the only start state q_0
$A := \emptyset$ ▷ A is the set of accepting states in $\mathscr{A}$
$R := \emptyset$ ▷ R is the set of rejecting states in $\mathscr{A}$
for each timed sample $\tau = (a_1, t_1), (a_2, t_2) \dots (a_n, t_n)$ from S **do**
  $q := q_0$, $i := 0$
  while $i \leq n$ **do**
    if $\nexists(q, q', a_i, [n, n']) \in \Delta$ for any q' and any $[n, n']$ **then**
      $Q := Q \cup \{q'\}$ ▷ Create a new state q'
      $\delta := (q, q', a_i, [t_{min}, t_{max}])$ ▷ Create a new transition δ
      $\Delta := \Delta \cup \{\delta\}$
    $q := q'$, $i := i + 1$
  if $\tau \in S_+$ **then** $A := A \cup \{q\}$
  if $\tau \in S_-$ **then** $R := R \cup \{q\}$
$F := A \cup R$
return $\mathscr{A}$

ting transitions according to an evidence value. The only drawback in this scenario is the possibility that a TAPTA may contain inconsistent states, i.e., states which are accepting and rejecting simultaneously. That happens because in a top-down approach the time information is not considered at construction time since all the delay guard span from the smallest delay (denoted as t_{min}) to the biggest delay (denoted as t_{max}) observed into the training set. Therefore, two strings with the same symbolic part but different timed parts, one accepting and the other rejecting, end in the same state.
RTI follows the top-down approach when building the TAPTA. At this stage, it allows for inconsistencies: later, all of them will get fixed by splitting the transitions. Listing 4 shows the TAPTA construction algorithm.

Algorithm 5 Splitting a transition: split

Require: A DRTA $\mathscr{A}$, a transition $\delta = (q, q', a, [n, n'])$, a time value $t \in [n, n']$, and a training set S
Ensure: δ is split at time t and $\mathscr{A}$ is updated
$\Delta := \Delta \setminus \{\delta\}$ ▷ Remove δ from $\mathscr{A}$
Remove the tree rooted in q' from $\mathscr{A}$
$Q := Q \cup \{q_1\} \cup \{q_2\}$ ▷ Add states q_1 and q_2 to $\mathscr{A}$
$\delta_1 := (q, q_1, a, [n, t])$
$\delta_2 := (q, q_2, a, [t + 1, n'])$
$\Delta := \Delta \cup \{\delta_1\} \cup \{\delta_2\}$ ▷ Add transition δ_1 and δ_2 to $\mathscr{A}$
Set q_1 as the root of $\mathscr{A}_1 := \text{tapta}(S^{\delta_1})$
Set q_2 as the root of $\mathscr{A}_2 := \text{tapta}(S^{\delta_2})$

The splitting operator is one of the characterizing traits of RTI. A split operation of a transition δ at time t divides the region of the DRTA into two parts. The first part is reached by the timed strings that activate δ with a delay value lesser than t or at most equal to t. The second part is reached by the timed strings that activate δ with a delay value greater than t.
In order to understand the exact result of a split, it is required to define the concept of suffix. Given a transition $\delta = (q, q', a, [n, n'])$, any timed string can be written as $\tau(a, t')\tau'$ where:

- τ is the prefix before activating δ.
- (a, t) is a pair composed of a symbol a, i.e. the label of δ, and a time value t satisfying the delay guard of δ, i.e. $n \leq t < n'$.
- τ' is the suffix after activating δ.

A suffix of $\tau(a, t')\tau'$ for δ is the timed stirng $\tau^\delta = (a, t')\tau'$. A suffix is said to be positive if $\tau\tau^\delta$ is positive, and it is said negative when $\tau\tau^\delta$ is negative. The set S^δ denotes the subset of the training set S containing all the suffixes from S for δ, i.e. $S^\delta = (S^\delta_+ = \{\tau^\delta | \tau \in S_+\}, S^\delta_- = \{\tau^\delta | \tau \in S_-\})$. It is also called suffix-set. As shown in Listing 5, the two parts of the DRTA divided by a split are reconstructed using these suffix-sets. That is made possible because the only splittable transitions in RTI are those pointing to blue destination states, which are roots of TAPTA subtrees by construction.
A final remark on the splitting operator regards its capability of removing inconsistencies. Suppose S^δ contains two suffixes τ^δ_1 and τ^δ_2 having the same symbolic but different timed parts. In the TAPTA, those two suffixes end at the same state because of how the TAPTA is constructed. After splitting the transition δ, it is possible that τ^δ_1 and τ^δ_2 no longer end in the same state. If τ^δ_1 and τ^δ_2 introduce an inconsistency, the split operator could fix it.

As shown in Listing 6, the merging operator in RTI is similar to its corresponding in State Merging. The only difference is where the delay guards are handled to avoid non-determinisms. RTI only merges red states with blue states and only merges when those states are compatible, i.e., accepting states and rejecting states cannot be merged. However, even when two states selected for for merging are compatible, their respective transitions may introduce a non-determinism due to some overlapping delay guard. In those cases, the standard determinization routine would solve the non-determinism by merging all the involved states. That behaviour would hinder the learning process even because the uncoloured states of the automaton have a high chance of introducing non-determinism since their guards span from t_{min} to t_{max}, i.e. they cover the largest interval possible. To avoid this problem, RTI splits the transitions in the blue state to fit the guards of the red state and does that before moving those transitions to the newly created merged state. Doing so allows the actual merging and the subsequent determinization procedure to continue with the same logic as in State Merging.

Now that all the principal ingredients have been discussed, it is possible to explain the RTI algorithm. Once the TAPTA has been constructed and the tree's root has been coloured red, RTI tries all possible merges, splits, and state colouring at every iteration. RTI col-

Algorithm 6 Merging two states: merge

Require: A DRTA $\mathscr{A}$, two states q, q' with q' not red, a training set S
Ensure: q, q' are merged, $\mathscr{A}$ is updated, and true is returned. Otherwise, false is returned

```
Q := Q ∪ {q''}                                   ▷ Add a new state q'' to 𝒜
if q ∈ A or q' ∈ A then A := A ∪ {q''}           ▷ A is the set of accepting states in 𝒜
if q ∈ R or q' ∈ R then R := R ∪ {q''}           ▷ R is the set of rejecting states in 𝒜
for each δ = (q', q*, a, [n, n']) ∈ Δ do
    if n' ≠ t_max then split(𝒜, δ, n', S)
for each δ = (q1, q2, a, [n, n']) ∈ Δ do
    if q1 ∈ {q, q'} then q1 := q''
    if q2 ∈ {q, q'} then q2 := q''
while ∃δ(q'', q1, a, [n, n']), δ(q'', q2, a, [n, n']) s.t. q2 ∉ Red do
    e := merge(𝒜, q1, q2, S)
    if e is false then
        Undo the merge between q and q'
        return false
return true
```

Algorithm 7 Checking permantent inconsistency: inconsistent

Require: A DRTA $\mathscr{A}$, and a training set S
Ensure: Returns true if $\mathscr{A}$ can still be made consistent with S. Otherwise returns false

```
for each q ∈ Red do
    if q ∈ A and q ∈ R then return false
    for each (q, q', a, [n, n']) ∈ Δ s.t. q' ∉ Red do
        if S+^δ ∩ S−^δ ≠ ∅ then                  ▷ (S+^δ, S−^δ) is the set of suffixes of S for δ
            return false
return true
```

lects an evidence value for each of those operations and chooses to operate, achieving the highest value[4] if and only if that operation does not introduce a permanent inconsistency.
A DRTA is permanently inconsistent if any of the following conditions hold:

- An inconsistency occurs in the red states.
- There exists an identical pair of positive and negative suffixes in the transitions to blue nodes.

The first condition cannot be fixed because red states are finalized and never updated by definition. The second condition cannot be fixed because there is no way a split can pull those suffixes apart. The permanent inconsistency check described in Listing 7 is invoked by RTI to avoid those pitfalls.

[4] In case of multiple operations having the same evidence value, merges are prioritized over splits, and splits are prioritized over colourings.

Algorithm 8 Identifying DRTAs: RTI

Require: A training set $S = \{S_+, S_-\}$ with alphabet Σ and having minimum delay value t_{min} and maximum delay value t_{max}
Ensure: $\mathscr{A}$, a DRTA consistent with S
  $\mathscr{A} := tapta(S)$
  $Red := Red \cup \{q_0\}$ ▷ Red is the red core of $\mathscr{A}$
  while $Q \setminus Red \neq \emptyset$ **do**
    for each $(q, q', a, [n, n']) \in \Delta$ s.t. $q \in Red$ and $q' \notin Red$ **do**
      $Blue := Blue \cup \{q'\}$ ▷ $Blue$ is the blue fringe of $\mathscr{A}$
    for $\delta = (q, q_b, a, [n, n']) \in \Delta$ s.t. $q_b \in Blue$ **do**
      for each $q_r \in Red$ **do**
        merge($\mathscr{A}, q_r, q_b, S$)
        if inconsistent($S, \mathscr{A}$) = false **then**
          Calculate the merge evidence value v_m
        Undo the merge between q_r and q_b
      for each suffix $\tau = (a, t)\tau' \in S^\delta$ **do** ▷ S^δ is the set of suffixes for δ
        split($\mathscr{A}, \delta, t, S$)
        Calculate the split evidence value v_s
        Undo the split of δ
      $Blue := Blue \setminus \{q_b\}$, $Red := Red \cup \{q_b\}$
      if inconsistent($S, \mathscr{A}$) = false **then**
        Calculate the coloring evidence value v_c
      $Blue := Blue \cup \{q_b\}$
  if a merge has the highest evidence value v_m **then**
    Redo the merge
  else if a split has the highest evidence value v_s **then**
    Redo the split
  else
    Redo the colouring with the highest evidence value v_c
return $\mathscr{A}$

A final remark on RTI is about the evidence values. As mentioned, RTI is based on the EDSM algorithm and computes evidence values to guide the decision on which operation to perform on the current hypothesized automaton. Indeed, RTI may be considered a greedy heuristic algorithm. RTI can use the standard EDSM evidence measure, which is based on the consideration that good merges are those causing, during the determinization procedure, the highest number of merges between accepting states and the highest number of merges between rejecting states. A merge achieving the highest EDSM value has been tested the most; therefore, there is the strongest support for being a good merge. However, besides the standard EDSM evidence measure, the authors designed and evaluated three additional measures, including or excluding the temporal information [81].

3

From Data to Messages

This chapter is based on a published paper [96].

This chapter starts the first part of the thesis and focuses on the data abstraction problem. When learning Automata from data, we must address the problem of translating them into symbols. This transformation aims to provide a higher-level data view according to preponderant features. The data abstraction is usually accomplished as a pre-learning step: before the automata inference algorithm starts, the data are translated into symbols. However, during the data abstraction, we might lose local dependencies among the data - i.e. a variation in a feature caused by a previous occurrence of some other feature. That may happen because a pre-processed data abstraction aims to assign the symbols according to a global overview of the data.

This chapter is based on a published paper [96] where the author's principal contribution was to introduce, for the first time to our knowledge, an automata learning algorithm that doesn't require an abstraction of the data into a symbolic domain.
Indeed, in this chapter, we consider the implications of postponing the data abstraction at learning time by proposing an algorithm that integrates automata inference and data abstraction.

The chapter is structured in four sections. Section 3.1 provides the reader with the context of the data abstraction problem. Section 3.2 introduces Regression Automata to model continuous signals. Section 3.3 presents RAI, Regression Automata Identifier, a novel algorithm that postpones data abstraction at learning time. Section 3.4 proves by experiment that the models inferred with RAI are competitive with those inferred with the standard methods, which requires human intervention to guide the data abstraction process.

3.1. INTRODUCTION

In this chapter, we investigate the problem of learning a structured model from a sequence of real-valued numbers or time-series. Such a model provides an interpretable visualization of the process underlying the sequence, see, e.g., [97, 98]. Furthermore, it can be used to make predictions about possible the value of future numbers. Our work is inspired by recent approaches for learning automata from sensor data [99, 100], multivariate sequences [101], event logs [102, 103], network traffic [104, 105], program specifications [106], and infinite alphabets [96]. The typical approach used when learning from such data is to first discretize any continuous values into discrete events. Sequences of such events are then provided as input to a learning algorithm for finding an automaton that could have generated this transformed data. This approach has two important drawbacks. Firstly, by separating the discretization from the learning process, it is hard to evaluate the quality of the discretization. Although the discretization clearly influences the learned model, we know of no measure that captures this influence. Secondly, this separation loses any influence the learned model has on the discretization. Structural models such as automata model a system using states, which determine the possible future sequences in the system. Frequently, different values are possible in different states. It is difficult to discover this dependence when performing a global discretization for the entire system beforehand.

In this chapter, the focus is on learning an automaton model from a univariate continuous signal while avoiding the aforementioned drawbacks. In particular, we develop a new test for state similarity, based on nearest neighbors [107]. We learn the automaton structure by computing the nearest neighbors of future sequences. Intuitively, when two states are similar, every future in one state is equally likely (depending on occurrence numbers) to find its nearest neighbor in the futures of its own and the other state. In this case, the states will be merged (combined) by the automaton learning algorithm. Similarly, a transition will be divided into two parts when the resulting two sets of futures share few nearest neighbors. In this way, our new learning algorithm called RAI (Regression Automaton Identification) learns transition guards. These guards are the discretization of the continuous valued signal into the discrete symbols. Instead of learning these beforehand, RAI learns them on-the-fly while learning the structure. The learning process thus directly depends on the discretization and vice-versa.

The result of RAI is a new kind of automaton, called regression automaton. A regression automaton is a deterministic model, i.e., the current state of the model is uniquely determined for any possible past sequence of real-values numbers. This makes these models easy to interpret. In addition, every state in a regression automaton contains a prediction value, e.g., the mean values of the first future number. These values can be used for time-series prediction: simply follow the path form the start state to the current state using the past symbols, and return the prediction value. We evaluate our novel model and algorithm on a synthetic data-set of a noisy sinus wave signal and on network flow records from the Zeus botnet. Both these signals are cyclic and deterministic, making them ideal use-case for testing whether the learned models are able to capture this behavior. To highlight also some limitations of RAI, we also test on very noisy time-series data coming from sensors measuring wind-speed.

We compare RAI with frequently used time-series models: ARMA, ARIMA, and continuous-valued HMMs. In addition, we compare the performance with the RTI+ algorithm for learning probabilistic real-time automata [108], which can also discretize the data on-the-fly. Our results show that RAI is able to capture the behavior in the deterministic signals into succinct interpretable models and achieve very competitive predictive performance. RAI outperforms all the common time-series models on the deterministic signals, and is outperformed on the noisy time-series data. Compared to RTI+, RAI is competitive in the sense that it frequently provides the better performing model. Overall, RTI+ is able to learn models with smallest prediction error. Showing that, although we developed several methods to adapt it, the nearest neighbor statistical test employed by RAI sometimes still makes wrong inferences. We believe the new test is very interesting because, in contrast to RTI+, it can easily be extended to multivariate sequences. All that is needed is the ability to compute distances between futures.

3.2. REGRESSION AUTOMATA

We introduce Regression Automata (RAs) as state representations of real-valued sequences.

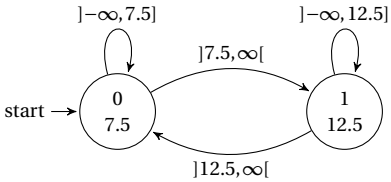


Figure 3.1: An example of a RA. The leftmost state is the start state. Each transition has a guard expressed as an interval. Each state contains an identification number on the top, and the predicted value in that state - $P(q)$ - on the bottom.

Definition 1. A Regression Automaton is a quadruple $\langle Q, q_0, \Delta, P \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the start state, Δ is a finite set of transitions, and $P : Q \to \mathbb{R}$ is a prediction function assigning a prediction value to each state in Q. A transition $\delta \in \Delta$ is a triple $\langle q, q',]l, r] \rangle$ where $q, q' \in Q$ are respectively the source and destination states, $]l, r]$ $l, r \in \mathbb{R}$ is a guard.

A first property of RAs highlights one of the peculiarities of this type of automata: they deal with an infinite domain, namely $\mathbb{R}$. In contrast, most of the existing automaton models as Deterministic Finite State Machines or Mealy Machines work on a discrete set of symbols.

Property 1. Given a RA $\langle Q, q_0, \Delta, P \rangle$, all the transitions leaving a state $q \in Q$ entirely cover $\mathbb{R}$ with their guards: $\forall q \in Q, \bigcup_{\langle q, q',]l, r] \rangle \in \Delta}]l, r] = \mathbb{R}$.

A second important property of RAs is that they are deterministic. An automaton model is deterministic when there is a unique state execution for a given sequence. RAs are deteministic because it is not allowed to contain transitions with the same source state and overlapping guards. Determinism in automata is a desirable property since the problem of learning non-deterministic automata is more complex than the problem of learning their deterministic counterparts - which is already NP-hard by its own [109]. Moreover, deterministic models are significantly easier to interpret because they can only be in one state at a given point in time. We refer to [61] for a detailed study on the complexity of learning automaton models.

Property 2. Given a RA $\langle Q, q_0, \Delta, P \rangle$, all the transitions leaving a state $q \in Q$ have non-overlapping guards. Formally: $\forall q \in Q, \nexists \delta_1 = \langle q, q'_1,]l_1, r_1] \rangle, \delta_2 = \langle q, q'_2,]l_2, r_2] \rangle, \delta_1, \delta_2 \in \Delta, \delta_1 \neq \delta_2$ s.t. $]l_1, r_1] \cap]l_2, r_2] \neq \emptyset$.

Suppose we provide the sequence $s' = [3, 12]$ to the RA in Figure 3.1. From the initial state 0, and observing 3, transition $\langle 0, 0,]-\infty, 7.5] \rangle$ is fired and we land in state 0 again. After observing 12, and still being in state 0, transition $\langle 0, 1,]7.5, \infty] \rangle$ is fired and the we reach state 1. Similarly, when sequence $s'' = [20, 8]$ is provided as input, transition $\langle 0, 1,]7.5, \infty] \rangle$ is fired at first, and then from state 1 and observation 8 transition

$\langle 1, 1,]-\infty, 12.5]\rangle$ is fired.

Definition 2. A computation of the RA $\langle Q, q_0, \Delta, P\rangle$ over a finite sequence of observations $s = [o_1, o_2, \ldots, o_n] \in \mathbb{R}^n$ is a finite sequence $q_0 \xrightarrow{o_1} q_1 \xrightarrow{o_2} q_2, \ldots, q_{n-1} \xrightarrow{o_n} q_n$ such that for all $1 \leq i \leq n$ $\langle q_{i-1}, q_i,]l_i, r_i]\rangle \in \Delta$, where $o_i \in]l_i, r_i]$.

Currently, RAs can be used as acceptors and predictors. They are not generative. Similar to Hidden Markov models (HMMs), they can be extended to include distributions such as Gaussians in every state. We leave such extensions for future work. For now, RAs simply predict the mean values of such distributions for any real-valued input sequence. For instance, sequence s' of the previous example predicts the value in state 1: 12.5. Formally, we define the prediction of RAs as follows.

Definition 3. The prediction for a given computation $q_0 \xrightarrow{o_1} q_1 \xrightarrow{o_2} q_2, \ldots, q_{n-1} \xrightarrow{o_n} q_n$ of the RA $\langle Q, q_0, \Delta, P\rangle$ is the $P(q_n)$.

Consider state 0 in the RA of figure 3.1 and suppose to have a sample

$$S = \{[6, 12, 11, 9], [4, 4, 6], [5, 8, 15]\}.$$

The observed future behavior of state 0 given S consists in all the (sub)sequences of S whose computation starts in 0. Since 0 is the start state, its observed future behavior given S is S itself. However, if we consider state 1, the observed future behavior is $S_1 = \{[12, 11, 9], [8, 15]\}$. We define the observed future behavior for a state $q \in Q$ as the tail-set for q given S, and all the (sub)sequences forming a tail-set as the tails of q given S.

Definition 4. The tail-set of state q from RA $A = \langle Q, q_0, \Delta, P\rangle$ given a sample S, is the set $S_q = \{o_i \ldots o_n$ s.t. $o_0, \ldots, o_n \in S$ and $q_0 \xrightarrow{o_1} q_1 \xrightarrow{o_2} q_2, \ldots, q_{i-2} \xrightarrow{o_{i-1}} q$ is a computation of $A\}$. The tail-set of a transition $\delta = \langle q, q',]l, r]\rangle$ is the tail set of state q'.

3.3. REGRESSION AUTOMATA IDENTIFICATION

Regression Automata Identifier (RAI) is inspired by the RTI+ algorithm for Real-Time Automata, discussed in [108]. It requires an input sample S composed of sequences of values in $\mathbb{R}$, a significance threshold $\alpha \in \mathbb{R}$, and a prefix length $p \in \mathbb{N}$. The input sample S is constructed from a long sequence of observations $s = [o_1, \ldots, o_N]$ using sliding windows, $S = [o'_i, \ldots, o'_{i+n}]$ where n is the window length [2]. The goal of RAI is to discover the regression automaton underlying S. RAI is a state merging algorithm using the red-blue framework [94]. As such, during its execution, RAI keeps two sets of states: a core of red states and a fringe of blue states. The red core represents that part of the automaton which has been defined and finalized, while the blue fringe represents that part of the

[2] There are several ways of estimating the window length. Usually, this parameter is set by the domain expert given her/his knowledge of the problem. In Chapter 7 we show how to compute the window length by means of autocorrelation plots.

automaton which has already been discovered but has yet to be finalized.

Listing 9 summarizes the algorithm. RAI builds a prefix tree that describes the sample S. RAI then starts merging pairs of states iteratively. By merging states, the model gains generality, i.e., it will express more behaviour than what is explicitly observed in S. Like RTI+ and early state merging methods such as Alergia [95], only merges between compatible red and blue states are allowed. Two states are considered compatible if the observed behaviour after reaching those states is similar. Essentially, this means that a Markov property holds after merging, i.e., the future is independent of the past given the current state. In RAI, the test for this property is performed using a nearest-neighbor-based statistical test. Like RTI+, the p-value of this statistical test is used as evidence to guide the selection of which states to candidates for merging. Section 3.3.3 discusses this consistency test.

If there are no compatible merges for a blue state, it is promoted to red. Blue states are created as children of the new red state. The transitions to these blue states are learned at this time by clustering observed futures together into guards. This clustering uses the same statistical test used for merging. RAI uses a bottom-up clustering strategy instead of the top-down strategy used by RTI+ to identify temporal guards. That means the guards are initialized to the smallest interval possible and eventually expanded by the clustering algorithm instead of starting as wide as possible and eventually being segmented into smaller parts.

Algorithm 9 RAI algorithm

Require: a sample of real valued sequences S, a significance threshold α, the prefix length p
Ensure: The result is a RA $\mathscr{A}$
  Construct the first p levels of prefix tree $\mathscr{A}$ from S, and color the start state q_0 of $\mathscr{A}$ red
  while $\mathscr{A}$ contains blue states **do**
    Select the red/blue couple (q_r, q_b) with the highest evidence score
    if q_r and q_b are consistent with confidence α **then**
      Merge q_r and q_b
    else if There are no compatible merges with q_b **then**
      Color q_b red and identify the guards for q_b outgoing transitions

3.3.1. PREFIX TREE CONSTRUCTION

A prefix tree is a direct translation of the input sample S into an automaton. In traditional state merging, every unique prefix sequence in S leads to a unique state in the prefix tree. For real-valued sequences, the only overlapping prefix sequences are those with identical values. A prefix tree in RAI is, therefore, a list of state lists connected by transitions with singleton guards. This is different from RTI+, where a prefix tree is built using guards that span the entire input range.

In addition to this difference, RAI only generates the first p levels of the prefix tree. The

reason is that in order to compute the nearest neighbour statistical test, we need to compute distances between futures. These futures are represented using tails. By generating only the first p levels of the tree and by setting the p parameter as discussed in Section 3.4, every state is guaranteed to contain tails whose length is at least $n-p$, where n is the sliding-window length. To avoid different distance computations due to different tail lengths, RAI compute distances only between tails of size $n-p$.

Figure 3.2 shows an instance of prefix tree generated by RAI with $p=2$. As it is possible to see, it only contains prefixes of length two of the strings in the input sample. The rest of those words are stored as tails for each state (not shown in the Figure).

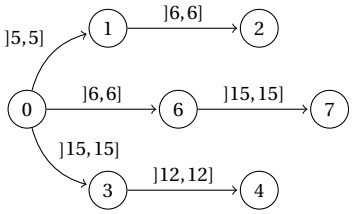
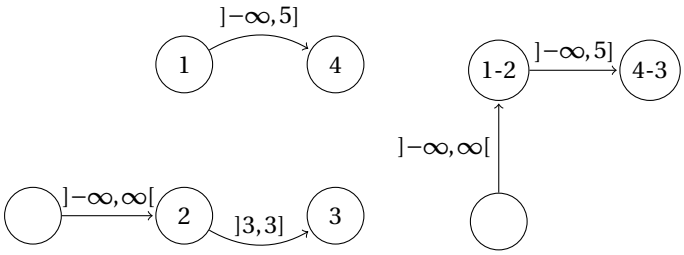


Figure 3.2: Prefix tree for sample $S = \{[5.5, 6, 15, 12], [6, 15, 12, 5.3], [15, 12, 5.3, 6.1]\}$ and prefix size $p = 2$.

3.3.2. MERGE OPERATOR

In RAI, we use a standard merge operator, see, e.g., [61]. Although merging may seem complicated due to the transition guards, it is not because of the red-blue framework. Suppose a red state q and a blue state q' are selected for being merged. Due to property 1, the guards of all outgoing transitions from q span the entire value range $\mathbb{R}$. In addition, all outgoing transitions of q' are controlled by singleton guards since q' is blue. Therefore, we can safely remove q' from the automaton, point its single incoming transition to q, and move all tails from q' to q. Then, we recursively merge every state q'' reached after q' with the state reached by the transition from q containing the singleton value of the guard on the transition to q''.



(a) Before the merge. (b) After the merge.

Figure 3.3: Example of merging operation between state 1, red, and state 2, blue.

As an example, consider the situation depicted in Figure 3.3. State 1, red, has been selected to be merged with state 2, blue. All the incoming transitions for state 2 are directed

to state 1. All the tails of state 2, not shown in the figure, are added to the tail-set of state 1. State 2 has a single outgoing transition with guard $]3,3]$ to state 3, by definition. Since state 1 is red, there exists a unique outgoing transition from state 1 such that it includes the singleton value 3: the transition from state 1 to state 4 with guard $]-\infty,5]$. Therefore, the operator merges state 4 with state 3 .

3.3.3. NEIGHBORHOOD-BASED STATISTICAL TEST

RAI uses a statistical test as evidence value to determine which pair of states to merge. Intuitively, for every pair of states, the futures can be seen as draws from two distributions. When we cannot reject the hypothesis that these two distributions are different, a merge of the two states is considered consistent. Amongst all consistent state pairs, like RTI+, we merge the pair with the largest p-value. This is the pair displaying the most similar futures, and hence we believe them to represent a single state in the system behavior. An input parameter is used to set the significance threshold.

Unlike RTI+, we use a test based on distances and nearest neighbors. Recall that S_q denotes the tail-set for state q, i.e., the set of all tails from the sample S leaving state q. We denote the kth nearest neighbor of a tail t in a tail-set Γ with $NN_k^{\Gamma}(t)$, computed using a metric $d : \Gamma \times \Gamma \to \mathbb{R}$. We define the following indicator:

$$I_k^{q,q'}(t) = \begin{cases} 1, & \text{if } NN_k^{S_q} = NN_k^{S_q \cup S_{q'}} \\ 0, & \text{otherwise.} \end{cases}$$

Therefore, the evidence value RAI uses is the following:

$$T_k(q,q') = \frac{1}{nk} \sum_{\tau \in S_q \cup S_{q'}} \sum_{r=1}^{k} I_r^{q,q'}(\tau),$$

where $n = |S_q| + |S_{q'}|$. In other words $T_k(q,q')$ is the proportion of all k nearest neighbor comparisons, after merging S_q and $S_{q'}$, in which a tail and its neighbor are members of the same tail-set S_q or $S_{q'}$. We expect a larger proportion when the two states represent two distinct states in the system bahvior. In that case, their futures will be different and hence their tail-sets dissimilar. We perform the following statistical test on this statistic.

Theorem. Let $n_q = |S_q|$ and, $n'_q = |S_{q'}|$ with $q, q' \in Q$. If $n_q, n_{q'} \to \infty$ with n_q/n tending to λ_q and $n_{q'}/n$ tending to $\lambda_{q'}$, then $\frac{\sqrt{nk}(T_k(q,q')-\mu_k)}{\sigma_k}$ has a limiting standard normal distribution under the hypothesis that S_q and $S_{q'}$ are similar, where $\mu_k = \lambda_q^2 + \lambda_{q'}^2$ and $\sigma_k^2 = \lambda_q \lambda_{q'} + 4\lambda_q^2 \lambda_{q'}^2$. [107]

The metric used by RAI to compute the nearest neighborhood is the Euclidean distance. However, as already discussed in section 3.3.1, the Euclidean distance is affected by the

size of the tails. In other words, when having two tails of different size, any decision on how to cope with missing values introduces a bias, e.g., by privileging shorter tails over the longer tails or vice-versa. To avoid this problem, RAI ensure to always compute the metric on equally sized tails. That is main reason why RAI limits the construction of the prefix tree to a certain number of levels. Since all tails have at least $n-p$ values as prefixes, we use these to compute the metric

$$d_p(t_q, t'_q) = \sqrt{\sum_{i=0}^{n-p} (t_q^i, t_{q'}^i)^2}$$

with $q, q' \in Q$, $t_q \in S_q$, $t'_q \in S_{q'}$, and t^i denoting the i-th value of tail t.

This distance computation should work fine in many settings, but can lead to unexpected results when learning RAs from sliding window samples. For instance, assume we have the sample $S = \{[5.5, 6, 15, 12], [6, 15, 12, 5.3], [15, 12, 5.3, 6.1]\}$ which has been generated by sliding a window of size four. Furthermore, assume we have $p = 2$. Consider states 0 and 1, whose tail-sets are $S_0 = S$ and $S_1 = \{[6, 15, 12]\}$. Notice that

$$d_p([6, 15, 12, 5.3], [6, 15, 12]) = 0,$$

therefore all the tails of state 1 have their nearest neighbor in S_0. Furthermore, the same happens with almost all the other states of the prefix tree since the root state contains all the tail prefixes of all the tails in its tail-set. This sliding window effect, causes RAI to assign higher scores to merges with the initial state.

To mitigate this problem, we proposes to use a slightly modified statistical test in the case of sliding windows. Assume to have states q and q', we partition S_q and $S_{q'}$ in two subsets $S_q = \{A, B_q\}$ and $S_{q'} = \{B_{q'}, C\}$. A contains the tails that occur in S_q and not in $S_{q'}$. B_q contains the tails of S_q that also appear in $S_{q'}$ due to the sliding window effect. Similarly, $B_{q'}$ contains the tails of $S_{q'}$ that also occur in S_q, and C contains the tails of $S_{q'}$ that only occur in $S_{q'}$. When considering states q and q' for merging, RAI applies Schilling's statistical test on A and $S_{q'}$ rather than S_q and $S_{q'}$. If A is empty, meaning all tails in S_q have the closest neighbor in $S_{q'}$, the statistical test is applied between $S_{q'}$ and C. Essentially, RAI neglects all the tails that are present due to the sliding window effect.

3.3.4. GUARDS INFERENCE

Like RTI+, RAI is able to learn transition guards on-the-fly, directly after promoting a blue state to red. RTI+ identifies the temporal guards of a DRTA (Deterministic Real Time Automaton) using a top-down approach, comparable to the divisive strategy in hierarchical clustering (see, e.g., [110]). This approach has the advantage of making clustering decision globally, using all the available data. This advantage usually comes with high computational effort. RAI adopts a bottom-up approach, which corresponds to an agglomerative strategy for hierarchical clustering. Essentially, any clustering method could be used once implemented. It may even be possible to link the learning of transition guards to a machine learning platform, as is done in [103] for global clustering of data

into symbols.

RAI initially sets the guards to cover just one value. These are extended by joining pairs of similar transitions. The advantage of this approach is that it is fast because it only considers pairs of candidates in each decision. The disadvantage is that it does not consider the global structure of the data, possibly leading to suboptimal decisions. Since RAI aims to learn transition guards, it sorts the outgoing transitions from the new red state by their singleton guard value. It then only considers joins (merges) between pairs of consecutive transitions. By doing so it cuts the space of all possible joins for a transition to only two: the predecessor and the successor in this order. The decision about which transitions to join first is made by looking at the evidence score computed on the candidate transitions tail-sets. The idea here is the same as for the merging criterion: RAI joins them if and only if there is sufficient evidence that the future behavior is similar regardless of which transition is fired.

3.4. EXPERIMENTS

We evaluate RAI on four different time-series forecasting tasks named *Sinus*, *Distorted-Sinus*, *Stratosphere*, and *wind*. Each task is composed of ten different test cases, and each test case consists of two time-series: a training set of 1000 data points and a testing set of 250 data points. Prediction provided by any model are evaluated by means of the Mean Absolute Error (MAE). Both the code used in our experiments, and the data, are available and free to share [111].

	RTI+ symbols	RTI+ time	ARIMA	ARMA	HMM	RAI	Persistence
Sinus	0.276, 0.278	0.284, 0.281	0.535, 0.623	0.54, 0.668	0.305, 0.302	0.28	0.81
Distorted-Sinus	0.365, 0.356	0.317, 0.459	0.578, 0.581	0.761, 0.732	0.327, 0.312	0.31	0.69
Stratosphere	6.088, 6.056	5.311, 9.699	191.2, 184.7	94.5, 94.6	73.953, 73.958	11.39	94.29
Wind	0.595, 0.787	0.537, 0.649	0.44, 0.425	0.679, 0.658	0.515, 0.533	0.83	0.34

Table 3.1: Average MAE for each technique (columns) and for each experiment (rows). When two valued are provided, they refer to different configurations of the sliding window length. In Stratosphere experiment, window sizes are 20 and 40, window sizes are 8 and 16 in others.

	RTI+ symbols	RTI+ time	ARIMA	ARMA	HMM	RAI	Persistence
Sinus	6	3	0	0	0	1	0
Distorted-Sinus	1	2	0	0	0	7	0
Stratosphere	1	4	0	0	0	5	0
Wind	0	0	0	0	0	0	10

Table 3.2: The number of times the technique (columns) is the best perfoming one for an experiment (rows).

3.4.1. SETUP

Time-series in *Sinus* experiment represent a sinus wave where each value is sampled in a different quadrant with uniform probability - i.e. in the first quadrant $\sin(x)$ is computed with $x \in [0, 90^o]$, in the second quadrant $\sin(x)$ is computed with $x \in [90^o, 180^o]$, and similarly for the other quadrants. The result of this sampling is a suite of 20 series having a period of four, and a randomly picked initial quadrant.

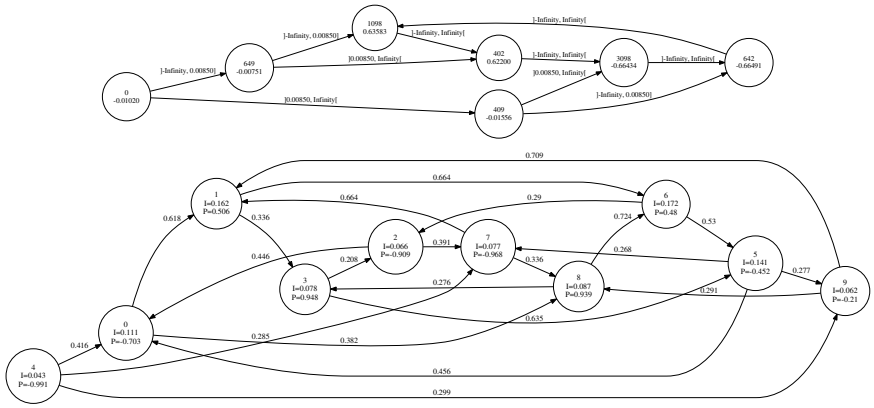


Figure 3.4: RA inferred by RAI (top), HMM inferred with Viterbi (bottom), on one of the Sinus experiments. In the RA, the sinus wave can clearly be observed in states 1098, 402, 3098, and 642. In the HMM, the wave is much harder to see. A high probability loop can be found over states 0, 1, 6, and 5, but none of these show the actual predictions made using the entire model as these are averaged over many states.

Distorted-Sinus experiment is the same as Sinus but before the sampling starts, a randomly value t within the second quadrant is selected as a threshold; every time there is a need for sampling in the second quadrant of the sinus, if the sampled value is bigger than t then the period is reset to the first quadrant. This experiment tests a test a situation where the best discretization depends on the system state.

In *stratoshpere* experiment there are time-series of delays in the time of arrival of consecutive netflows. A netflow [65] is an aggregation of network packets that share a common key. The key is a 5-tuple comprised of the source IP address, source port, destination IP address, destination port, and protocol. The netflows we considered in our experiment come from a network capture from Zbot, also known as Zeus botnet, a known distributed malware [112]. As discussed in [113], those data show a clear periodicity.

Wind experiment is about wind speed prediction, similar to [100]. The data come from sensors located in Rotterdam, the Netherlands, and consist of five minute spaced average wind speed observations [114] To generate the series we preprocess the data by aggregating them on a hourly base, and we select ten random and non-overlapping intervals of 1250 data points. The first 1000 data points are used for training, the remaining 250 for testing.

Since RAI uses only half of a sliding window (using the p paramter) for learning the RA structure, and half only for estimating futures, we provide two different window sizes for the other methods. Important is to set a window size that is longer than the period of a cycle, otherwise an automaton learning technique will generate a tree-shaped model. On Sinus, Distorted-Sinus, and Wind experiments we used sizes 8 and 16 (and only 16 for RAI, with p values 8). On Stratosphere, we use windows of size 20 and 40.

3.4.2. ALTERNATIVE TECHNIQUES

We compare RAI with other six alternative techniques: *RTI+symbols*, *RTI+time*, *ARMA*, *ARIMA*, *HMM*, and *Persistence*. *RTI+symbols* is an instance of the standard methodology for learning automata from continuous signals we discussed in section 3.3. It consists of translating the training series into symbolic sliding window samples. On the Sinus and Distorted-Sinus dataset the signal is modeled by using an alphabet of two symbols, one representing positive values and the other representing negative values. On Stratosphere we designed an alphabet of three symbols: the first one ranging from 0 to 23 milliseconds of delay, the second ranging from 24 to 349 milliseconds of delay, and the third one ranging from 350 milliseconds of delay and above. On the Wind experiment we have used the same setting of [100], resulting in an alphabet of eight symbols. We refer to that work for the interval boundaries.

It is important to notice that after having learned an automaton with RTI+symbols, it is necessary to estimate per-state predictors since RTI+ target models does not have the capability of emitting predictions. The estimation procedure consists in executing all the windows within the sample with the learned automaton, and for each state collect all the values reached that state along the execution. The predictor in each state will be the mean of those values.

RTI+time is an alternative technique that, as RAI, does not need the discretization step. RTI+ learns real time automata, which capture temporal properties in the form of delay guards. RTI+ also requires symbolic information representing events in the system. We remove this need by providing RTI+ with a 0 symbol for all events, and representing the continuous signal using the time values. This technique uses the same prediction function estimation as RTI+symbol.

ARIMA and *ARMA* [115, 116] are two widely used techniques for time-series forecasting. In each experiment we train ARIMA and ARMA models by setting the autoregressive order parameter to the window size, and the moving average order to 1 (since the cyclic behavior is important, not the trend).

Hidden Markov Models (*HMM*) [117] are one of the most widely used statistical models. In our experiments we used HMMs with Gaussian emissions, since they are appropriate for modeling continuous-valued signals, $ad10$ components. HMM learning is achieved by applying Viterbi algorithm on the sliding window samples with at most 1000 iterations.

Persistence is a widely used technique in time-series prediction. Persistence always predicts the previous value, and despite of its simplicity it is considered as a strong baseline.

3.4.3. RESULTS

Table 3.1 shows the average MAE along the ten test cases in each experiment. Furthermore, Table 3.2 presents the best performing technique in all the test cases of each experiment and all window sizes. RTI+symbols performs the best when given the perfect

discretization in the Sinus signal. This is to be expected, as the learning problem is then only to discover the 6 state model shown in Figure 3.4. The guards in Figure 3.4 are not perfect however, since these are identified by RAI based on noisy data. This slight imperfections in guard values exlains the slightly worse predictive performance of RAI and RTI+time compared to RTI+symbols. All other time-series models perform worse, although the difference with HMMs is small. As can be seen in Figure 3.4, the learned HMM model is much harder to interpret due to its non-deterministic nature.

On the Distorted-Sinus and Stratosphere experiments, RAI achieves the best performance overall. The RTI+ result in terms of MAE are better then RAI. This is because the new RAI statistic still draws some wrong conclusions, leading to large models with large MAE. RAI does find the best performing model 5 out of 10 times. This experiment also shows the benefit of learning guards on-the-fly, as RTI+time and RAI both perform very good. Furthermore, none of the time-series methods is able to accurately capture the cyclic behavior in the Stratosphere data. Lastly, as the Wind experiment shows, RAI is not very good at modeling non-cyclic noisy signals. As is shown in [100], using automata on such data requires additional preprocessing.

On the Sinus experiment with window size 16, the learning times on a 64 bit laptop with intel Core i7 processor and 8GiB RAM, are: 0.159 for RTI+symbols, 0.198s for RTI+time, 4.429s for ARIMA, 12.402s for ARMA, 153.919s for HMM, and 12.402s for RAI. RTI+ and RAI not only perform well, they are also very efficient in terms of run-time.

4

FROM STREAMS TO STRINGS

This chapter continues the journey through the several critical problems that arise when learning Automata from streams: the data segmentation. Applying the techniques discussed in the previous chapter makes it possible to obtain an abstraction of the data in the form of a stream of symbols. However, all the automata inference algorithms can only learn meaningful models by starting from a sample of sequences of symbols called strings. This requirement introduces the necessity of producing strings from a stream of symbols. The most used method to address this problem is sliding a fixed-size window over the stream and returning one string for each window. However, the sliding window method may cause the introduction of unwanted and unseen behaviour in the target automaton.

Section 4.1 of this chapter discusses the sliding window method in detail. The sliding window approach comes with the drawback of introducing some noise within the inference process - namely strings that should not be characteristic of an automaton. Later in the chapter, we propose a novel approach which aims to find particular locations along the stream where to segment it. As discussed in Section 4.2, indication of where to segment the stream is obtained by automatically expanding prior knowledge in the form of few cut-locations provided by a supervisor. A big space is reserved for the experiments, which are the principal topic of Section 4.3.

4.1. SLIDING A WINDOW

The research on Grammatical Inference has produced several algorithms to learn probabilistic automata. Those tools rely on a provided sample of strings called sample. However, there are cases where such a sample is not directly available, and only a stream of symbols is available. We know those symbols form strings produced by some hidden model, but we do not know where are the boundaries between consecutive strings. Once the words have been detected, any state of the art learning algorithm may be used to learn a probabilistic stateful model. An often used approach is about segmenting the stream according to a given criterium, usually based on heuristics belonging to specific application contexts. In network data processing, for example, streams are cut using thresholds on the interpacket time distances.

When a domain knowledge is absent, the most used approach is to set a sliding window on the whole stream to identify sequences of the same fixed size. Such a technique comes with different advantages like the simplicity of implementation, but it turns out to have no theoretic foundations and sometimes its usage is only motivated by the lack of alternatives. Given a stream of symbols S and a size k, the sliding window (SW) approach will produce exactly $|S| - k + 1$ words moving a fixed size window along S. If $S = baaabbbba$, and $k = 3$, the following words: $baa, aaa, aab, abb, bbb, bbb, bba$ will be generated. The listing 10 shows how to make a stream of words from a stream of symbols using a sliding window approach.

In this work we focus on learning a particular type of probabilistic automaton called Probabilistic Deterministic Automaton (PDA), defined in the next section.

Algorithm 10 Sliding window string generator

Require: stream S of unsegmented words, and window size k
  $W \leftarrow \epsilon$, $w \leftarrow \epsilon$
  for each symbol $s \in S$ **do**
    if $|w| < k$ **then**
      append s to w
    else
      add w to W
      remove the first symbol from w
      append s to w
    end if
  end for
  add last string w to W
  return W

4.2. SEMI-SUPERVISED SEGMENTER

Assume to have a stream S composed of a concatenation of strings produced by a Probabilistic Deterministic Automaton (PDA). The PDA, also referred as the target automaton, is unknown and the location of the boundaries between the strings is unknown as well. However, let assume to know a small set of strings produced by the target also called knowledge base. The proposed method, the Semi-Supervised Segmenter (SSS), leverages the knowledge base to segment the stream with the goal of learning a PDA which is as close as possible to the actual target. It is worthwile to highlight that SSS does not concern the learning of the PDAs, since this task is achieved by state of the art algorithms developed of this purpose [95, 108, 118]. The goal of SSS is to generate a training sample from the stream by making those algorithms able to learn PDAs which are closer to the target than the ones learned by using other methodologies like the sliding window.

SSS relies on a language model to make a decision about placing a boundary along the stream. A language model is a probability distribution on all the words belonging to a language. Language models are a widely used tool to solve speech recognition, word segmentation for asiatic texts, part-of-speech tagging, and other natural lnguage processing tasks. Ngram models are one possible mean to implement language models. Ngrams are contiguous sequences of n items, in this case strings taken from S. The length of the ngram is called order, and if the order is one the model is called unigrams, if the order is two then the model is called bigrams, just to cite few common instances. One of the most common techniques to estimate the probabilities of ngrams is by maximum likelihood estimation. A maximum likelihood estimator (MLE) assigns probabilities by maximizing the likelihood given the stream S. With $s, s' \in S$, we have:

$$P(s') = \frac{\#(s')}{\#_S}, P(s) = \frac{\#(s)}{\#_S}$$

$$P(s'|s) = \frac{\#(s \oplus s')}{\#(s)}$$

Where $\#(\cdot)$ is an operator denoting the amount of times its argument has been observed in S, $\#_S$ denotes the amount of strings in S, and the $\oplus$ operator denotes the concatenation of its arguments.

However, as mentioned at the beginning of this section, we do not know where the boudaries between consecutive strings are on S. Therefore, SSS extracts the strings from S by sliding two consecutive windows; those windows generate couples of ngrams having a pre-defined maximal order k and represeinting contiguous strings. As an example, with $S = baaabbbba$ and $k = 3$ SSS extracts (ϵ, baa), (b, aaa), (ba, aab), (baa, abb), (aaa, bbb), (aab, bbb), (abb, bba) with ϵ denoting the empty string. SSS counts how many times each ngram has been observed after the previous ngram within the knowledge base and it leverages those counts to estimate probabilities by maximum likelihood. SSS estimates probabilities for ngram models of order 1, 2, and 3.

After the estimation, SSS scans the stream looking for new boundary locations. By using the double sliding window as at counting time, now it is able to compute the probability of observing two contiguous strings. Given two consecutive strings $s, s' \in S$, SSS explores all the possible suffixes $\mathscr{S}_s$ of s and all the possible prefixes $\mathscr{P}_{s'}$ of s' to find the couple which minimizes the sum between the probability of placing a boundary after a suffix and the probability of placing a boundary before a prefix:

$$\min_{(\rho,\sigma)\in\mathscr{P}_{s'}\times\mathscr{S}_s} \{P(\epsilon|\sigma) + P(\rho|\epsilon)\}$$

SSS uses this quantity as an evidence for placing a boundary between s and s', i.e. the bigger it is, the higher is the confidence.

The reason for exploring all possible suffixes of the former string and all the possible suffixes of the latter string is because it needs to consider eventual strings whose size is shorter than 3, and that is also the reason why it leverages multiple length ngrams.

Listing 11 summarizes the SSS algorithm, with $MLE(B,3)$ denoting the maximum likelihood estimation procedure for the ngram models of orders 1 to 3 given the knowledge base B, with $DSW(S,3)$ denoting the sequence of consecutive trigrams along S, and with $P_{NG}(\rho|\sigma)$ denoting the ngram probability of observing ρ after σ.

4.3. EXPERIMENTS

Evaluating how good an automaton is may be a hard task for several real-life cases such as the one mentioned in the introduction. In botnet Command and Control (C2) channel analysis, for instance, a big problem is often the lack of an actual C2 state machine specification. Infected hosts belonging to the same botnet adopt the same communication protocol implemented by the botnet designer, which in most cases is the only person who knows it. The lack of a C2 protocol state machine hampers the evaluation since there is no ground truth to compare with.

That is the main reason why we decided to plan experiments on a set of generated models. The PAutomaC [109] suite is composed of 48 probabilistic models of different dimen-

Algorithm 11 Semi-Supervised Segmenter

Require: a stream S of unsegmented words
Require: a set of strings B as a knowledge base
Require: the number of boundaries to place t
  initialize T as an empty list
  $NG \leftarrow MLE(B,3)$
  for each $s, s' \in DSW(S,3)$ **do**
    $dmin \leftarrow \infty$
    for each $\rho \in \mathscr{P}_{s'}$ **do**
      for each $\sigma \in \mathscr{S}_s$ **do**
        $d \leftarrow P_{NG}(\epsilon|\sigma) + P_{NG}(\rho|\epsilon)$
        if $d < dmin$ **then**
          $dmin \leftarrow d$
        end if
      end for
    end for
    store $(s, s', dmin)$ in T
  end for
  place a boundary between s and s' having the top t scores in T

sions and features already used in the past for a learning competition. Table 2 in the appendix reports many of those characteristics such as the number of states, the alphabet size, or the model type. Each PAutomaC problem comes with an exhaustive description of the model: a training set composed of a list of strings emitted by the model and used for learning purposes, a test set consisting still of strings meant for evaluating purposes, and a solution file containing the probabilities assigned to each word in the test set.

The primary goal of the evaluation consists in assessing the quality of several techniques for generating a training sample from a stream of concatenated strings. The quality is a property of the model we learn from that training sample, and it is an indication of how close is the inferred model to the actual target model. Since the target models in our experiments describe probability distributions over strings, by using them, it is possible to assign a probability to each string within the test set. After having normalized those probabilities, we use the perplexity given the solution to assess the quality of a candidate model:

$$Perplexity(TS) = 2^{-\sum_{x \in TS} P_G(x) \times \log(P_C(x))}$$

Where TS is the test set, $P_G(x)$ represents the probability assigned to the word x by the correct model, and $P_C(x)$ represents the probability assigned by the candidate model to the word x. Perplexity has been chosen because of its capability of assessing prediction quality regardless of the model type and because it is the same evaluation measure used in the PAutomaC competition.

4.3.1. SETUP

For each problem in the PAutomaC suite, we use the original evaluation set containing 1000 strings and the corresponding solution to keep the possibility of comparing with the PAutomaC competitors. Furthermore, for each problem, we use the original model to generate a sample of 4000 strings. We decided to generate a new training set instead of using the original in the suite to exclude the empty strings and still have the same amount of data to learn from in all the problems. By concatenating the strings of a training set, we obtain the stream of concatenated strings, which will be the input of all the alternative techniques evaluated in this work.

The first technique considered in the experiments is the *sliding window* with windows of five symbols. A second technique is the *random*, which consists of segmenting the stream on 3999 randomly chosen locations called boundaries. We repeat the selection ten times, collecting ten random segmentations for each problem. Furthermore, we have *partially-random* segmentations with a percentage β of the 3999 correct boundaries and $(1-\beta)\times 3999$ randomly picked boundaries. We boost β incrementally by adding 5% and starting from a minimum of 5% up to 100%. Thus, we create 19 partially-random configurations with increasing share of the correct bounds. Also, as for the random technique, we repeat ten times the selection of the boundaries for each configuration, generating a total of 190 semi-supervised segmentations for each problem. Finally, we include the *semi-supervised* segmentations obtained by adopting the algorithm described in section 4.2. The knowledge base is created by randomly selecting a percentage of the exact boundaries for the correct part of the partially random segmentations, with the difference that, in this case, the algorithm will place the rest of the boundaries instead of choosing them by random. As for the partially random segmentations, we have an increasing knowledge base by incrementally adding 5% of the right boundaries, and we repeat their selection ten times, generating 190 semi-supervised segmentations.

Each segmentation is a training sample since each segment is a string. We use those training samples to learn PDAs with the ALERGIA algorithm discussed in Chapter 2.[1] In the next section, we discuss the quality of the learned models.

4.3.2. DISCUSSION

Table 5.4 gives an overview of the quality of the models learned by ALERGIA on several training sets. As already mentioned, we measure the quality of a candidate model using perplexity. Since we know the correct target model, we gather the probabilities assigned to each string in the evaluation set by both the candidate and the target. We compute the perplexity of the candidate probabilities given the target probabilities, after a mandatory normalisation step.

Table 4.2 shows all the different techniques, discussed in section 4.3.1, to obtain a training set starting from a stream of concatenated strings. For two of them, namely the partially-random (PRN) and the semi-supervised (SSS), it shows different parametrisations. SSS technique is instantiated with a knowledge base composed of the same per-

[1] We used the ALERGIA implementation included in the Treba learning tool [119].

centages of correct boundaries of the former technique. It is important to mention that given the same β, the correct boundaries used in both PRN and SSS are precisely the same. Furthermore, the perplexity of all the techniques reported in the table, with the only exception of the sliding window (SW) which is deterministic, is averaged on ten runs having each one a different random seed.

It is interesting to observe that the sliding window represent a viable option since it is better than the random baseline on 33 over 48 PAutomaC problems. Also, it achieves the best perplexity score on 7 out of 48 problems among all the considered alternatives.

With only 200 correct bounds, corresponding to $\beta = 5\%$, SSS is already better than SW on 33 problems, and better than random on 35 problems. With 600 correct bounds, corresponding to $\beta = 15\%$, SSS shows a lesser perplexity than SW on 36 problems: an amount that we consider as significant. Also, SSS outperforms the random alternative on 38 problems. We believe that, in case of network data, it is possible to isolate 600 sessions by hand and use that knowledge base to learn less perplex models.

It is interesting to consider the comparison between SSS and PRN. By fixing β, SSS_β outperforms PRN_β 32 over 48 times with $\beta = 5\%$, 33 over 48 times with $\beta = 15\%$, and 36 over 48 times with $\beta = 25\%$. Figure 4.1a shows the perplexity scores of the considered techniques for a specific problem (20) by incrementing β. As it is possible to observe, 400 exact boundaries ($\beta = 10\%$) are enough for SSS to outperform the other alternatives. However, on some problems, the number of correct bounds required by SSS may be unfeasibly high. When considering the minimal percentage of correct bounds, also referred as β_{min}, we list 11 problems where $\beta_{min} > 50\%$ to have SSS models achieving the lowest perplexity. Table 4.1 shows them and provides for the corresponding β_{min}. Figure 4.1b shows the perplexity plot with increasing β for one of those instances (39).

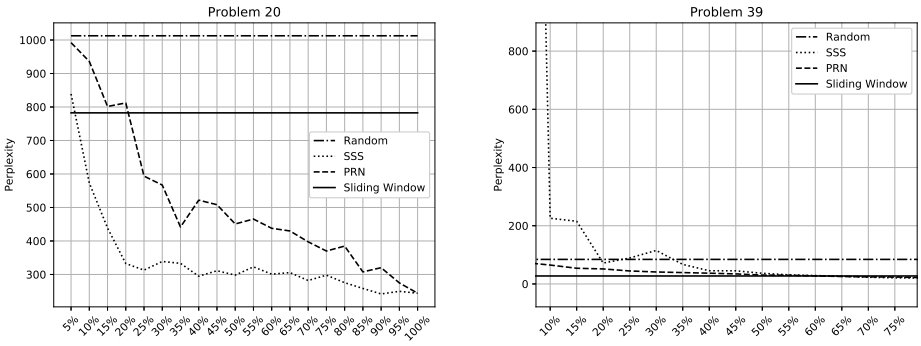
By considering the meta-informations about the PAutomaC problems reported in Appendix B (table 2), we have evidence about SSS struggle with some instances characterised by a non-deterministic nature and low transition sparsity.

Table 4.1: Semi-supervised segmenting (SSS), the proposed technique, needs β_{min} percent of the correct bounds to enable the inference of the less perplex model. On a total of 48 problems of the PAutomaC suite, in 11 cases β_{min} is significantly high - i.e. $\beta_{min} > 50\%$.

ID	3	6	15	30	31	32	38	39	41	42	43
$\beta_{\mathbf{min}}$	100%	100%	90%	100%	100%	100%	100%	70%	100%	100%	100%

Table 4.2: Overwiew of the quality of the models learned by ALERGIA on several training sets. Those sets are generated by using the following techniques: sliding window, random, partially-random (PRN), and semi-supervised (SSS). Each perplexity value is averaged on ten runs with different random selection of the boundary locations (with the exception of the sliding window technique which is deterministic). For PRN and SSS there are thee instantiations having $\beta = 5\%, 15\%, 25\%$. The lowest perplexity values for each problem are in evidence.

ID	Sliding Window	Random	$PRN_{5\%}$	$PRN_{15\%}$	$PRN_{25\%}$	$SSS_{5\%}$	$SSS_{15\%}$	$SSS_{25\%}$
1	581.289	985.559	939.566	897.197	811.674	428.510	**329.354**	374.668
2	**680.049**	1295.562	1294.280	1270.563	961.421	1271.512	686.473	861.090
3	260452.615	1271.570	1248.737	1198.033	**1173.126**	4495.283	3570.932	2775.250
4	>600000.0	15819.115	10110.851	7918.812	6263.169	67185.747	13944.509	**2564.822**
5	>600000.0	890.599	912.830	908.540	809.091	304.451	276.364	**213.270**
6	558699.988	34485.959	23366.728	15937.556	**13524.776**	205270.824	181215.184	30801.969
7	460917.042	1744.499	1862.626	1549.534	1585.937	43984.432	805.142	**637.015**
8	72553.689	136432.228	115044.406	115141.441	69354.809	11269.890	**10353.172**	11475.293
9	>600000.0	18373.695	10818.599	16808.595	5679.396	1194.627	1062.913	**715.887**
10	2274.343	5368.656	5138.007	4596.014	4441.754	2011.304	1647.213	**1563.046**
11	53546.777	81076.713	79826.128	73368.257	72873.098	20838.543	**18209.727**	20477.023
12	2559.463	4077.790	2855.919	1959.365	1273.696	1553.705	**1106.227**	1273.041
13	>600000.0	20429.808	18212.446	16075.721	12370.256	19325.681	13310.475	**10520.326**
14	370.914	437.869	370.730	327.333	**323.072**	367.448	376.973	424.743
15	**811.020**	2015.400	1737.012	1454.424	1249.453	3072.511	2512.938	1891.455
16	1399.228	1860.830	2023.016	2023.503	1975.102	**804.016**	996.372	993.539
17	4694.467	4937.499	4686.294	3317.353	2395.770	4752.315	2241.294	**1545.891**
18	3145.812	12384.465	12634.710	5169.935	4235.565	3993.860	2851.223	**1937.452**
19	176.705	397.463	349.591	258.248	175.719	104.324	**90.522**	96.380
20	782.492	1012.620	992.070	801.544	593.609	838.817	438.747	**312.999**
21	9240.274	20153.174	20685.811	13011.105	12274.275	9507.766	**4934.344**	5982.294
22	2210.926	6355.885	3660.252	2317.583	1899.118	**488.253**	493.496	624.674
23	**118.989**	225.713	216.117	220.166	199.838	200.237	148.125	120.034
24	73078.781	5858.792	4465.322	3892.953	3731.848	6121.332	1739.571	**597.956**
25	2347.801	2907.830	3018.525	2768.083	2411.972	**1495.089**	1620.670	1876.009
26	292656.726	295529.213	219066.633	142365.751	112897.943	134238.107	105863.802	**104876.725**
27	11529.481	19411.150	18446.562	19020.908	15190.707	**3438.447**	5828.930	5873.876
28	1859.168	1425.095	1361.719	1112.942	1143.819	1362.171	984.243	**708.859**
29	5698.217	2217.498	2032.550	1709.192	1520.088	3521.239	1427.658	**1303.701**
30	408.681	594.794	557.978	466.271	**362.496**	2441.556	547.635	519.213
31	681.597	643.509	622.119	562.073	**494.779**	1786.108	867.800	683.644
32	5197.070	821.159	811.798	682.560	**640.752**	1057.716	722.197	682.950
33	127.170	215.808	214.729	147.162	**76.720**	118.620	105.346	99.630
34	5452.433	5905.622	5934.085	5934.597	5929.767	3705.255	3608.233	**3309.362**
35	7529.175	17081.711	11984.568	8652.235	6628.976	**1443.567**	2114.052	1826.611
36	415.187	584.544	569.420	564.882	552.034	415.828	**350.558**	372.178
37	66.445	424.118	379.644	375.558	329.559	60.440	60.228	**58.206**
38	138.918	59.609	65.701	57.378	**53.182**	108.356	81.170	82.114
39	**27.407**	84.370	73.982	53.906	44.635	4221.407	215.326	89.215
40	59.691	75.725	75.808	71.372	67.854	**46.322**	50.275	51.086
41	**28.344**	425.117	416.851	311.374	260.699	32.231	30.211	33.888
42	**35.667**	98.648	85.550	71.247	65.153	159.484	96.110	95.376
43	**95.661**	494.671	535.318	529.654	520.731	122.046	124.744	127.937
44	69.271	68.324	68.384	67.196	66.196	48.887	**45.874**	51.668
45	120.479	234.346	206.550	199.243	171.279	129.344	**100.089**	106.524
46	54.556	60.533	58.191	56.277	48.197	**42.490**	65.030	49.727
47	94.147	58.666	44.623	31.664	25.577	17.906	17.433	**17.281**
48	146.740	162.760	164.314	155.431	148.095	**112.707**	115.970	149.035



(a) (b)

Figure 4.1: Perplexity plot with increasing β for PAutomaC ptoblem 20. On that instance, $\beta = 10\%$ of the total correct boundaries in the training stream of concatenated strings, corresponding to 400 strings, are sufficient for the semi-supervised technique to outperform the other alternative techniques (a).
Perplexity plot with increasing β for PAutomaC ptoblem 39. In that case the semi-supervised segmenter (SSS) requires 70% of the total correct boundaries in the training stream of concatenated strings, corresponding to 2800 strings (b).

5

Collecting Data for Automata Learning

This chapter is based on a published paper [120].

This chapter completes the first part of the thesis by addressing the problem of deciding the minimal amount of data to collect to learn a meaningful automaton. As far as we know, this topic still needs to be addressed in the automata inference research community. The main contribution of the chapter consists of introducing a signal that may help the human estimate when the collection of further data isn't contributing to adding new behaviour modelled in the target automaton.

This chapter is based on a published paper [120] where the author's principal contribution was defining the data abstraction technique, consisting of the labeler and sequencer components described in the introduction chapter. The data abstraction technique is a prerequisite for learning the communication profiles, namely the automata employed for determining if sufficient data has been gathered.

The content is organized as follows: Section 5.1 provides the reader with the context of the data collection problem. Section 5.2 introduces our solution for building communications profiles using finite state machines. Section 5.3 and 5.4 present experiments and their evaluation. Finally, we discuss the evaluation results in Section 5.5.

5.1. INTRODUCTION

Network traffic monitoring is of paramount importance for efficient network management. It allows for timely intervention in case of any observed failures, intrusions or changes in network communication patterns [121]. Initially, network traffic analysis relied primarily on inspecting network packet contents. Due to the high volume of data exchanged in modern networks, this in-depth analysis of the whole traffic is no longer realistic. A more common approach is to analyze aggregated communication information of which IP flow records is an example. The main challenge in using this data lies in the extraction of relevant information from this meta data. In this paper, we focus on the problem of creating a model to classify hosts based on their traffic summary statistics. We refer to this task as behavioral communication profiling. The problem of current methods addressing this task is the use of batch processing techniques over large amount of data inducing delay in model learning due to long period of data collection. Processing large amount of data limits the complexity of the analysis methods due to space and computation limitation. Consequently, these simple methods are not able to model accurately communication profiles.

To address these limitations we propose to use complex models for modeling fine grained communication profile with finite state machines. We use stream learning methods allowing us to build a communication profile in real-time as network traffic is observed. We show that the minimum amount of training data needed to learn an accurate communication profile can be determined on the fly. We assess that profiles learned from limited IP flow data are as efficient as ones using more training data for botnet detection.

Such fast communication profiling method has several applications such as real-time learning of malicious behaviors observed on honeypots or known infected hosts. It ensures that malicious profiles can be learned fast and applied as early as possible for de-

tection. Another application is the profiling of normal behavior for anomaly detection. Our method ensures to learn a normal profile as fast as possible while avoiding under-trained models that can raise false alarms.

5.2. BUILDING COMMUNICATION PROFILES

This section is structured in three parts. In Subsection 5.2.1, we introduce the concept of communication profile. Subection 5.2.2 discusses the data abstraction procedure that is required to learn communication profiles from data. Finally, Subsection 5.2.3 addresses the problem of collecting the miminum amount of data to learn meaningful automatons representing communication profiles.

5.2.1. COMMUNICATION PROFILES

A communication profile provides a concise description of a participant or a group of participants in a network. We build profiles only using connection-level communication information provided by IP flow records. The main task is to extract the key behavior from the records, and reduce the data into a compact description. Given IP flow records from an unknown source, we can classify; given a known source, we can predict future behavior. Mathematically, a communication profile is a PDFA which we learn from IP flow records as described in Chapter 2. To infer information about a single host from its IP flow records, we aggregate consecutive flows within a short time period into a single word and use a sliding window technique to obtain sequences of words describing consecutive flows. These words are descriptions of short-term behavior.

5.2.2. ENCODING IP RECORDS FOR PDFAS

IP flow records are tuples of features stated in Table 5.2. Because PDFAs take strings as input, we need to convert IP flow records from their representation to sequences of symbols. The product of all possible combinations of values for each feature, i.e. $|range(protocol)|\cdot\ldots\cdot|range(data_{rec}|)$, using a common discretization of time in ms and traffic in bytes, yields too many combinations to be practical. That would cause the learning algorithm to produce large models, and the behaviour described by those models would be too specific to be useful for classification or prediction. Common approaches to this issue are clustering and discretization. We follow the latter approach and map each feature value to a symbol describing its percentile.

Indeed, for each numerical feature f_i $(1 \leq i \leq d)$, we choose a finite number of bins b_{f_i}. For each of the $\Pi_{i=1}^{d} b_{f_i}$ combinations, we add a symbol to the alphabet Σ. Each value of the categorical feature is a signed number. This encoding has advantages over a standard clustering approach because symbols created via encoding tend to be easier to interpret for humans. Moreover, encoding avoids defining a metric to measure the distance between two IP flow records. To use, e.g. kNN clustering [122], a more appropriate metric than Euclidean distance needs to be used to derive good PDFAs based on words obtained from the clusters.

For example, assume to have flows with only two features

Protocol: Represents the transport protocol for all the packets summarized by a given flow. It is a categorical feature with two possible values: TCP and UDP. TCP is mapped to the value 0 and UDP to 1.

Duration: Represents the total duration of a given flow. It is a numerical feature with two percentile-intervals $[0,49]$ and $[50,\infty[$ mapped to 0 and 1, respectively.

Since each feature is mappable in two different symbols, there are four possible combinations of feature values. If we capture the sequence of flows

$$f = \langle TCP,5\rangle, \langle TCP,57\rangle, \langle UDP,2\rangle$$

then flow $\langle TCP,5\rangle$ is mapped to symbol $0\times\frac{4}{2}+0\times\frac{2}{2}=0$. Flow $\langle TCP,57\rangle$ is mapped to symbol $0\times\frac{4}{2}+1\times\frac{2}{2}=1$. Flow $\langle UDP,2\rangle$ is mapped to symbol $1\times\frac{4}{2}+0\times\frac{2}{2}=2$. Therefore f is mapped to the sequence of symbols (also said string) $s=0,1,2$.

After encoding the flow records, we aggregate all flows starting within a fixed time by sliding a window over the stream and incrementing the start of the window one flow at a time. Therefore a word consists of a sequence of symbols within a window, where each symbol represents a flow.

5.2.3. DATA ESTIMATION CRITERIA

The prefix tree automaton (APTA) is the starting point for all state merging learning algorithms. Because the tree is a compact way to represent all the training data, it offers ideal access to analyze the impact of varying training set sizes on the learning process. The key in minimizing the data needed to learn a model is understanding the error introduced by using a partial sample of the data: It enables us to analyse the quality provided by a partial view of the data with respect to the complete data. We apply two criteria to judge the completeness of the partial sample: For a formal approach, we check the Hoeffding bound, a type of concentration inequality [123]. It bounds, with high probability, the error made when estimating a function value calculated on a random source. For an informal, application-driven approach we observe growth in states and transitions when adding more data to the prefix tree. Equation 5.1 states the Hoeffding inequality. It bounds the difference between the true mean r of a random variable with range R with its estimation $\bar{r}$ calculated on a finite sample with low error δ: With probability $1-\delta$, the error in the estimation $\bar{r}$ only deviates by an ϵ from r. The *true mean* r is the mean calculated on all, possibility infinite samples.

$$r \leq \bar{r} - \epsilon \text{ with prob } 1-\delta \text{ where } \epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}} \tag{5.1}$$

The estimation is sub-linear in terms of the confidence δ and quadratic in sample number for precision ϵ. We apply this technique to the APTA by estimating the relative frequency $\frac{c_i}{n_s}$ of transition i in each state s where $n_s = \sum_{c_i \in s} c_i$. This allows us to bound the error in the empirical probability distribution defined by occurrence counts. We stop collecting more data once the error is below a given threshold δ.

Table 5.1: Summary of the malware IP flow record dataset published in [124]. Records are labeled as malicious, normal, or background traffic.

ID	#Flows / Duration / Size	Malware (#bots)	Class Distribution back / bnet / norm
10	5,180,852 / 4.75 hrs / 73GB	Rbot (10)	29.33 / 67.97/2.69
11	40,836 / 0.26 hrs / 5.2GB	Rbot (3)	29.33 / 67.97 / 2.69
12	1,262,790 / 1.21 hrs / 8.3GB	NSIS.ay (3)	96.98 / 2.34 / 0.68

5.3. EXPERIMENTS

We conduct two rounds of experiments: First, we measure the amount of data needed to reach various confidence thresholds in a prefix tree using the Hoeffding bound as well as the growth in number of states to judge prefix tree completeness. Second, we learn communication profiles from the training sets determined empirically in previous experiments. Considering IP flow records, it is not clear how to realize a 80:20 training to testing data split, which is commonplace in machine learning [122]. Is it correct to split after 80% of the observed packets, flows, bytes, or maybe after 80% of the time? Often, these features are not correlated and will yield wildly different training and testing sets depending on which feature is chosen as a measure.

The first experiment is designed to determine if, and how quickly, a full data representation can be obtained from a partial view of the data, as well as how large the generated training sets are. In a second experiment, we empirically validate reduced datasets obtained from the first experiment by learning communication profiles from the sets. We compare their performance in host classification with profiles trained on the full training set. Together, the experiments help to answer the question if, and how, we can determine the right amount of data to learn a finite state machine as a communication profile to classify hosts.

5.3.1. DATASET

We use a publicly available dataset of manually labeled IP flow traces [124]. The flows contain communication from hosts running botnet malware as well background and legitimate traffic. The dataset is organized in several scenarios (ID[2]), each running one or more infected hosts connected to the internet. Table 5.1 summarizes the dataset.

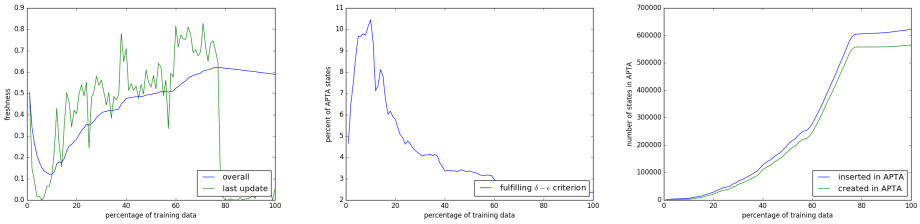
5.3.2. DATA PREPARATION

We encode the IP flow records using the features stated in Table 5.2. Numeric attributes are discretized by assigning a number according to the percentile its value is in. The percentiles themselves are obtained by selecting a random subset of IP addresses from the normal traffic to calculate the statistics. Any knowledge transfer is prevented by excluding these flows addresses from any further experiments. All flows irrespective of their duration, starting within $t = \tau\ ms$ are collected in a window to obtain short term interaction patterns of each IP address. We advance the window on a per-flow level. The

[2]The names are kept from [124]

Table 5.2: Features extracted from each IP flow per monitored host. Hosts of interest are filtered by IP address. The remaining fields are input for the learning algorithm. $Time$ is derived from the flow's starting time.

Features	Description	Values
$protocol$	transport protocol of the flow	categorical: tcp, udp, etc.
$time$	time since previous flow started	timestamp
$duration$	duration of the flow	time in ms
$pakets$	Count of packets exchanged	numerical
$data_{exc}$	Amount of data exchanged	numerical, in KB
$data_{rec}$	Amount of data received	numerical, in KB



(a) (b) (c)

Figure 5.1: Identification of bad data preprocessing using training set from Scenario 10, obtained with 4 bins on 250ms windows (Figure (a) and (b)). The blue line shows the overall freshness of the training data inserted to the APTA. The green line indicates the freshness within the last update containing 1% of the training data. Due to the long windows, a difference in a prefix of the window will lead to many new states. Despite the slight drop in freshness depicted on the left towards the end of the graph, fewer states fulfill the Hoeffding bound shown on the right. Likewise, depicted in (c), the same scenario with a shorter window of 50ms, but 10 bins per feature. In both cases, the training set is not big enough to learn meaningful communication profiles.

duration τ is chosen using the streaming data analysis.

5.3.3. STREAMING DATA COLLECTION

In our experiments, we observe two different criteria for stopping data collection: In an application-driven approach, we observe the "freshness" Δ of samples w_s with respect to an APTA A. We define it as the ratio $\frac{[\![w]\!]}{|A|}$ of number $[\![w]\!]$ of states newly created in APTA A when adding sample w versus the total number of states $|A|$ in APTA A. Here, $[\![\cdot]\!]$ denotes the length of the word w minus the length of its longest prefix in A. When w is a set, we define $[\![w]\!] = \sum_{w_i \in s} [\![w_i]\!]$ as the sum of states created from the samples in the set. Adding samples that are already contained or have large prefixes in the tree only adds little extra information. The freshness ranges between 0 and 1, and low values indicate that the sample already has many duplicates, or at least long prefixes in the APTA. It serves as an indicator: if it falls below a threshold, the prefix tree already contains most of the data. This measure does not guarantee good estimates of the transition probability in each state since it does not account for the occurrence counts of states. To account for that, we compare to an statistics-driven approach, commonly used in stream learning: empirical distributions in the states of the APTA have to be bounded by the Hoeffding bound with varying thresholds. The more states have distributions bounded, the bet-

ter the APTA summarizes the true source. The Overlap merge heuristic does not use the distributions defined in each state to make decisions. In contrast, Alergia uses the empirical distributions defined in states to be merged to calculate their similarity. The application-driven approach, only estimating the size of the prefix tree is enough to estimate the data for heuristics like Overlap, but not for Alergia. If duplicates or samples which already have long prefixes are added, the number of new states created while inserting into the prefix tree will go down. By putting a threshold on the growth rate of the prefix tree, we can find the point when adding more data will not contribute to the learning process. Other heuristics use local distributions, and data for these are better bounded by the Hoeffding inequality.

Table 5.3: Overview of the merge heuristics adopted when learning automata from data: for each pair of states, called merge candidates, a score is determined using a heuristic. The pair with the highest score gets merged, and the process is iterated on the resulting automaton. The stopping point is also determined by the heuristic.

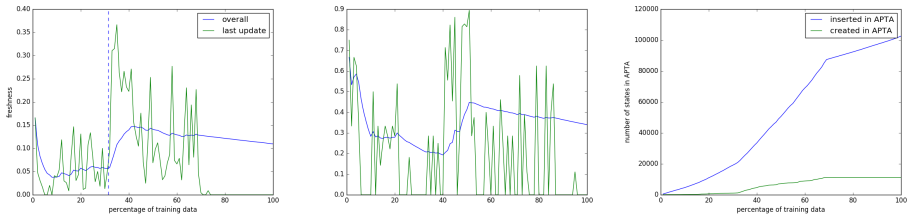
Heuristic	**Core Idea**
Alergia [125]	The heuristic calculates empirical transition probabilities in each state. Candidates are scored highly if distributions of outgoing transition labels match.
Overlap [126]	Identifies similar if-then-else flows: the more matching outgoing transition labels candidate states have, the higher the score. The heuristic ignores occurrence counts.

5.3.4. PROFILING BEHAVIOR

We use the dfasat software package [126][3] to learn communication profiles. The goal is to obtain a small automaton that can reliably distinguish legitimate from botnet sources. The classification task focuses on hosts, not individual traffic flows. We use the full training sets, as well as smaller training sets obtained from an analysis of freshness and Hoeffding bounds on local distributions to learn communication profiles. We use different heuristics, as outlined in Table 5.3. Because we assume a temporal connection between different IP flow records when building profiles over short windows of time, we only use a simple split into training and testing data instead of cross-validation or a shuffle-split approach.

To judge whether a host is malicious or not, we evaluate its associated communication profile, an APTA A, by calculating its acceptance rate of encoded IP flow records from an evaluation set. We discretize the flows in the evaluation set using the same method as for the preprocessing, and collect flows in windows of the same length. Since we want to reason about hosts based on their associated flows, we calculate the ratio of accepted versus rejected windows. A preliminary analysis infered that an acceptance ratio exceeding 75% any time after the first 25 windows is a good threshold to classify hosts as malicious.

[3] To reproduce our results, set threshold flags y and t to 1; to only use the heuristic set b to 1 and d to 2000. Run one iteration by setting n to 1. Turn sinks off by setting i to 0.



(a) (b) (c)

Figure 5.2: Overall freshness in Scenario 10 (a) and Scenario 12 (b). The blue line shows the development of the overall freshness, the green line depicts the freshness of the last update adding 1% of the training data to the prefix tree. The dashed vertical line indicates a point of change: local updates suddenly contain a lot of new samples without prefixes, or much longer samples. The last graph shows the number of states inserted vs the number of states created in the APTA in Scenario 10 (c). It shows that a lot of windows share prefixes.

5.4. RESULTS

In the experiments we first analyzed the training sets using various estimation criteria. Based on the analysis, training sets were in turn used to learn communication profiles.

5.4.1. STREAMING DATA COLLECTION

Figure 5.1 summarizes the behavior of our method for wrong choices during preprocessing. Long windows of 250ms or large alphabets with 10 bins per feature result in no convergence, neither in freshness nor in the percentage of transitions fulfilling the Hoeffding bound.

For the remaining experiments, we chose a small alphabet size obtained through few bins (4 per feature) and short windows ($\tau = 20\ ms$). An interesting observation across the different scenarios is the non-monotonicity of freshness. It clearly illustrates that the global behavior of a host is composed of several small, different behaviors. This property is captured by PDFAs, which can have multiple loops with transitions of high probability, connected by transitions of lower probability. This is particularly easy to see in Figure 5.2(a), indicated by a vertical dashed line: after adding increasingly less new information to the prefix tree, the updates at the 32% mark of the training set add a new behavior. The increase in freshness shows that words inserted encode behavior without prefixes in the APTA, i.e. previously unseen behavior. This is also visible in a plot of the states inserted into the prefix tree, i.e. the length of the samples, and indicates that windows start to contain more words. The dataset description of Scenario 10 lists a sequence of bandwidth increases and a switch from a UDP-based flood attack to an ICMP-based attack. The former did not use up the full bandwidth, the latter did. This makes extreme values and monotonicity of freshness an interesting candidate for clustering behavior.

Figure 5.3 shows the fraction of transitions fulfilling the Hoeffding bounds for a weak choice of parameters, $\delta = 15\%$ and $\epsilon = 0.15$. It shows that in neither scenario the ratio of transition bounded correctly exceeded 30%. Often, after initial going up, the ratio

Table 5.4: A summary of the baseline results, calculated on the whole training set for one infected IP address per scenario. There are 9 malicious hosts in Scenario 9 and 10, and 2 hosts in Scenario 11 and 12 each. The environment contains 48 benign hosts.

Experimemt	Alergia TP / FP / Pr	Overlap TP / FP / Pr
Baseline in Scenario 10	6 / 0 / 1	7 / 0 / 1
Baseline in Scenario 11	2 / 0 / 1	2 / 0 / 1
Baseline in Scenario 12	1 / 0 / 1	1 / 0 / 0.9
Experimemt	**Alergia TP / FP / Pr**	**Overlap TP / FP / Pr**
48% in Scenario 10	6 / 0 / 1	7 / 0 / 1
12% in Scenario 11	0 / 0 / 0	0 / 0 / 0
50% in Scenario 11	2 / 0 / 1	2 / 0 / 1
52% in Scenario 12	1 / 0 / 1	1 / 0 / 0.9

declines as more data gets added. Overall, this behavior is not surprising: States closer to the root are part of many prefixes, and therefore occur very often. As the prefix tree branches out, and adds more states per level, occurrence counts go down. The majority of states are close to the leaves of the tree, not to the root. This shows that decisions made by Alergia might not be reliable. Moreover, while the resulting state machine might accept and reject samples correctly, the associated probabilities might be off.

5.4.2. PROFILING BEHAVIOR

We use the training datasets determined in the previous step to learn PDFAs as communication profiles. The baseline is calculated on a communication profile trained on all IP flow records of one malicious IP address in each scenario. By inspecting the freshness, we chose 48% of Scenario 10, and 52% of Scenario 12 training data. For both cases, Figure 5.2(a) and 5.2(b) shows a plateau in global freshness, and the freshness of local updates is also low. In Scenario 11, freshness keeps increasing until the end, but is very low ($\Delta < 0.13$). We chose two splitting points: the low point of freshness at 12% of the training data ($\Delta = 0.03$), and for the lack of another extreme point, we also split at 50%. Table 5.4 summarizes the results for the most important metrics: true and false positives (TP/FP) and precision (Pr), a ratio of $\frac{TP}{TP+FP}$ describing how many of the identified hosts were relevant. For all but the 12% split, results for the communication profile learned from the reduced set are the same as from the baseline. It is very likely that the learning algorithm can infer the core structure from the reduced set and generalize enough. The inability to detect the malicious hosts in Scenario 11 with only 12% of the training data is not surprising. Just observing the freshness can be deceptive: a highly redundant representation of additional data can add valuable data to discriminate hosts. Without it, the communication profile rejects all data. Overall, the results are promising: We did not need to select features or tune parameters for different scenarios. Moreover, we only used word acceptance for evaluation, but the communication profiles also provide probabilities. We expect identification of malicious hosts as well as rejection of benign hosts to be better if we account for this information.

(a) (b) (c)

Figure 5.3: The Hoeffding inequality applied on transitions in the APTA, using $\delta = 15\%$ and $\epsilon = 0.15$ (on range $[0,1]$), depicted for Scenarios 10 (a), 11 (b), and 12 (c). Despite the rather weak parameters, there is not enough samples to fulfill it in a majority of states. As the freshness in Scenario 10 goes up (c.f. Figure 5.2 (a)), the fraction of transitions with Hoeffding bound starts going down around the 32% mark.

5.5. DISCUSSION

Our development and empirical study serves multiple purposes: Practically, it shows how to use a class of learning algorithms that has a high run time complexity with big data. Using a smart analysis of the dataset with the need of the algorithm in mind, it is easier to estimate how much data is really needed. Moreover, it helps to identify bad choices in preprocessing. This step is extremely important, as the effectiveness of the learning algorithm will depend on the choices like discretization and window length taken during preprocessing. The empirical study experiments does not change the established theoretical bounds in terms of complexity and data requirements, but we get a better understanding of the algorithm in our use case. In theoretical analysis, an assumed data generating finite state machine is to be recovered by identifying an automaton isomorphic to the generator. Complexity bounds typically are stated in terms of unknown quantities of the target, e.g. the number of states of the target automaton and the size of the alphabet. In our application, we don't know the former, and for the latter, the product of the number of bins chosen gives an upper bound for the alphabet size. Theoretical bounds derived from that are very conservative, and often prove to be impractical. The communication profiles we learn can be used in various ways, e.g. by setting up honeypots to learn malicious behavior directly from attackers. Profiles from these hosts can later be used to detect anomalies in network traffic.

6

Botnet Behaviour Fingerprinting with Timed Automata

This chapter is based on a published paper [105].

The third part of the thesis concerns how it is possible to leverage Automata and Automata inference for botnet detection. It includes this chapter and chapter 7. To our knowledge, we are the first to propose behavioural detection methods that rely on automatically inferred state machines to detect infections. Common network intrusion detection techniques usually rely on hand-coded rules, which may trigger an alarm when some network data meet them (i.e. lists of known IP addresses of malicious machines). Keeping those rules updated, adding new rules, and removing the rules which are not helpful anymore is a non-automated task performed by experts. With automata and automata inference techniques, it is possible to automatically generate rules matching a host's observed behaviour. In this chapter, we propose a technique which uses Timed Automata to describe a device's behaviour when communicating with other devices in a network. Furthermore, it allows the generation of behavioural features that can be used for detection. The use of time when describing network behaviours is debatable. On the one hand, timing is an essential part of the network data type, and some attacks, such as the Denial of Service Attacks, can be described with time categories. On the other hand, detection systems that rely on timed features may be circumvented by new-generation malware, which uses random delays in their communication protocols.

This chapter is based on a published paper [105] where the author's principal contribution was the first definition of a methodology for using automata automatically learned from data to detect compromised hosts in a network under observation. This methodology was later improved and refined, as discussed in the following chapter.

This chapter explores the possibility of including time information within the behavioural descriptions of the hosts.
Section 6.2 introduces the Probabilistic Deterministic Real-Time Automata (PDRTAs). Section 6.3 discusses the problem of how to automatically infer PDRTAs from network data and how to use them for detection purposes. Section 6.4 evaluates our PDRTAs-based detection system in different scenarios.

6.1. INTRODUCTION

Botnets pose a significant threat to cyber-security. Bots are zombie computers, remotely controlled by a malicious entity, and are used for attacks, spam, phishing and information exfiltration [127, 128]. Despite recent research, detecting and countering botnets is still considered an unsolved problem [121]. In Feily's survey [129], three categories of botnet detection methods are distinguished: signature-based [130], anomaly-based [131] and DNS-based [132, 133]. In signature-based detection, some characteristic like hashes are calculated, either on the malware binary, or from resource usage or from packet content capture. These characteristics serve as signatures used to identify the same malware or packets in the wild. Botnet developers counter this detection using techniques such as code obfuscation, encryption, and polymorphic code [134], makings signature-based detection increasingly ineffective. DNS-based detection techniques rely on anomalies in the DNS traffic, caused by the infected hosts' need to locate and communicate with a command and control server, which is usually hosted by a Dynamic DNS provider. This type of detection often wrongly considers hosts as malicious,

e.g., due to fake-domains and reconnaissance poisoning [129]. These false positives make DNS-based techniques fairly unreliable. The last category, anomaly-based detection methods are often behavior-based and monitor the run-time execution behavior of malware, which is much more difficult to conceal [135]. Consequently, there has been a large amount of research devoted to the development of effective behavior-based malware detection and analysis tools, see, e.g. [136, 137]. Behavior-based malware detection or analysis applies machine learning techniques in order to automatically learn models from data such as network traffic. In state machine learning, an instance of generative learning, is of particular interest: it can detect a botnet in new data, but its generative property also allows to infer and analyze the logical structure underlying the observed traffic. Depending on the data, in some cases it is even possible to infer a state machine diagram communication protocol used by the botnet, see [138].

In this chapter, we introduce BASTA (Behavioral Analytics System using Timed Automata), which uses probabilistic deterministic real-time automata (PDRTAs) to obtain identity fingerprints of hosts from timed network traffic streams. It is a behavior-based system, and learns models from Netflow traces instead of full packet contents. Packet content is typically used as information source to identify the basic event types/messages used in communication protocols. Netflows only contain information on the sources and targets of flows, the amount of data transmitted, the network protocol used, and the timing of the flows. This makes it much harder to infer a botnet's communication protocol. If successful, however, this approach opens up the road to many new applications of this technology because Netflow traces are widely available, while access to the content of messages is typically restricted due to proprietary or privacy related issues.

BASTA models specify behavior over timed events. We are especially interested in this timing information because it can be very important for determining network traffic behavior [139, 140]. In addition, PDRTAs can be learned efficiently from unlabeled data [108], making them an ideal candidate for modeling network traffic behavior. The development of BASTA is driven by the practical need of network administrators that run a wide network composed of many hosts that require monitoring. A common action taken after identifying an infection is a hard reset of the given machines in order to restore it to a trusted state. This operation is often an expensive one. In contrast to most Intrusion Detection Systems (IDS) that label individual packets/flows as suspicious, BASTA is focused on ranking hosts using an indicator of suspiciousness based on all of the outgoing and incoming flows. This indicator models the overlap in communication behavior between a given candidate host and a known infected machine. A low indicator means that frequencies of behavioral patterns observed in the entire set of flow records match the expected frequencies obtained from a known infected machine. A high indicator means that these frequencies do not match, and thus that the traffic shows no sign of this infection. Although we initially developed BASTA in order to detect such known infections, we demonstrate in this work that many of the patterns that it learns from Netflow records are generic: initial results demonstrate that BASTA is capable of detecting new infections from unknown malware.

In our empirical study on publicly available Netflow data, we obtain a high detecting rate, catching 100% of infected hosts in some scenarios, while maintaining a low false positive rate. These results are surprising, considering the low detection rates that have been reported using existing techniques on the same data [124]. Although the obtained results are not directly comparable due to the fact that these existing methods try to label every individual flow, our results do indicate that assigning labels to hosts rather than flows is a very promising direction for botnet detection. In addition, they show that timed automata are very effective tools for capturing the behavioral patterns in all of this data. In noised settings, the default settings used by BASTA can produce too many false positives to be used by network administrators directly. Since BASTA is a ranker of hosts, however, an admin can opt to only inspect the most suspicious ones, i.e., the hosts that are most likely malicious. Furthermore, since BASTA is a machine learning tool geared toward learning useful models from network traffic, it can be used as a replacement of standard machine learning methods used by malware detection frameworks such as for instance DISCLOSURE [141]. When using BASTA as a base detection system, such frameworks will continue to make use of many other tricks such as filters on IP range and port usage in order to further reduce the number of false alarms.

6.2. PROBABILISTIC DETERMINISTIC REAL-TIME AUTOMATA

PDRTAs are a probabilistic version of Real Time Automata [93], and they are probabilistic automata that include guards on the transition timings (inter-event) times. Formally, the events are modeled by timed strings $(a_1, t_1)(a_2, t_2)\cdots(a_n, t_n)$, where t_i denotes the time delay between the occurrences of the ith and $i-1$th events. The PDRTA model defines a probability distribution over such timed strings, having a Markov property in the distribution over events, and a semi-Markov property in the time guard.

Definition 21. A PDRTA is a 4-tuple $\langle \mathscr{A}, \mathscr{E}, \mathscr{T}, \mathscr{H} \rangle$, where

- $\mathscr{A} = \langle Q, \Sigma, \Delta, q_0 \rangle$ is a 4-tuple defining the machine structure: Q is a finite set of states, Σ is a finite set of event types (symbols), Δ is a finite set of transitions, and $q_0 \in Q$ is the start state;
- $\mathscr{E}$ and $\mathscr{T}$ are the event and time probability distributions, respectively. $\mathscr{E} : (Q, \Sigma) \to [0,1]$ returns the probability of generating/observing a given event in a given state. $\mathscr{T} : (Q, H) \to [0,1]$ returns the same but for a given time range $[v, v'] \in \mathscr{H}$, where $\mathscr{H}$ is a finite set of non-overlapping intervals in $\mathbb{R}_+$.

A transition $\delta \in \Delta$ in a PDRTA is a tuple $\langle q, q', a, [m, m'] \rangle$, where $q, q' \in Q$ are the source and target states, $a \in \Sigma$ is a symbol and $[m, m']$ is a temporal guard.

Figure 6.1 illustrates a PDRTA inferred from Netflow data. On the timed string $\langle Q - TCP, 500 \rangle \langle TCP, 50 \rangle \langle TCP, 200 \rangle$, it goes from state 1, via state 4 and 7, to state 2, its probability is computed as: $0.28 \cdot 0.21 \cdot 0.59 \cdot 1.0 \cdot 0.86 \cdot 1.0 = 0.03$. It can also represent a distribution over $(\Sigma, \mathscr{H})^*$ by adding final probabilities.

Note that in a PDRTA the states are defined by their event-time value distributions and

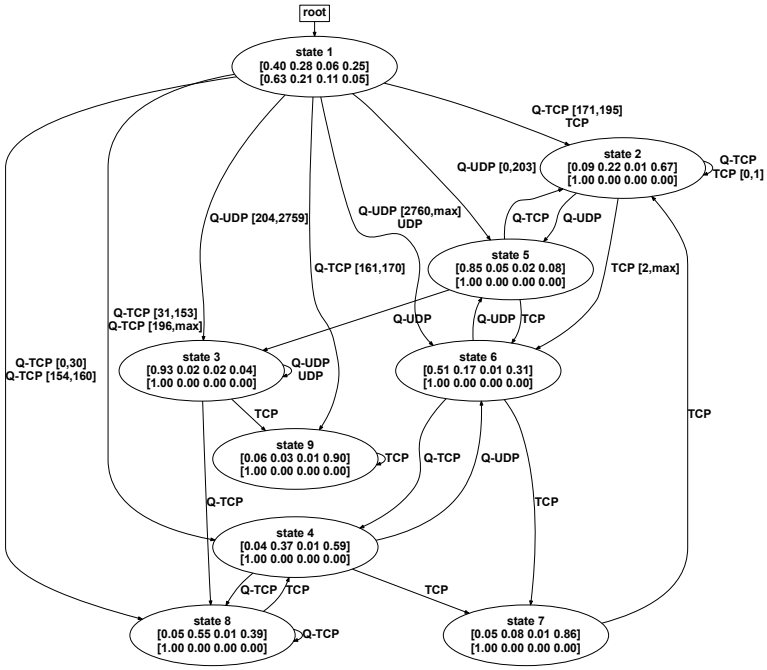


Figure 6.1: A PDRTA inferred from a sample of malicious Netflow traces in Scenario 9 of our dataset. States 2, 3, and 9 are spamming states. The botnet initiates many Quick (short duration) UDP/TCP flows. Edges are labeled with events and temporal guards. The latter are omitted if they are empty. The states contain two distributions: one for events, one for time intervals. The 4 event types in order: Quick UDP, Quick TCP, Other UDP, Other TCP. Breaks-points of the time intervals are: 485, 981, and 1660 ms.

their transitions to future states. These cannot be directly observed in data but have to be estimated using a learning method. PDRTAs are therefore similar to a timed variant of the HMM [81].

We learn PDRTAs instead of regular automata or Markov models because time information is important for characterizing network traffic. In PDRTAs, the influence of time values on the string probabilities is equal to that of all the other data contained in abstract events. Other types of stateful models such as Hidden Markov Models or Mealy Machines have been used for detection purposes [124, 142], but cannot be used to infer time constraints. We focus on deterministic automata because identifying non-deterministic automata is harder [61].

6.3. PDRTAs INFERENCE

We use a recent state machine learning algorithm, RTI+ [108] to learn malicious behaviors from Netflow data, and then use these as fingerprints for detection.

Here we briefly review this algorithm. RTI+ is based on the state-merging approach [94]. An untimed probabilistic state-merging algorithm starts by building a large tree-shaped automaton called prefix tree from a sample of input strings. Every state of this tree can be reached by exactly one untimed string and therefore encodes exactly the input sample. The algorithm then greedily merges pairs of states (q, q') in this tree, forming a smaller and smaller machine. When the target machine is deterministic, for every event $e \in \Sigma$ the states reached from q and q' have to be merged as well (the determinization process). By iteratively applying these merges, the algorithm generalizes over the sample and learns the structure of the target machine used to generate the sample. The algorithm uses a heuristic to decide merges and avoids overfitting. In an unsupervised setting, the merge heuristic is determined using statistics. A merge between q and q' is considered good if the future behavior after reaching q is similar to that after reaching q', which can be tested using, e.g., a likelihood-ratio test [108]. This essentially tests the Markov property, i.e., whether future behavior is independent of being in state q or q'. When these futures are significantly different, the merge is considered inconsistent and will not be performed. In addition to state merges, RTI+ is capable of performing transition splits [108]. In the prefix tree, the temporal guards include all possible time values. A split of a transition $\delta = \langle q, q', a, [m, m'] \rangle$ at time point t creates two new transitions $\langle q, q_1, a, [m, t] \rangle$ and $\langle q, q_2, a, [t+1, m'] \rangle$. The target states q_1 and q_2 are the roots of two new prefix trees that are reconstructed based on the input sample. In this way, RTI+ can learn temporal constraints in addition to the machine structure. For more details such as pseudo code and complexity analysis, the reader is referred to [81].

6.3.1. NETWORK FLOWS

Netflows are sequences of packets passing on a given network link, from a source host to a destination host. As such, they are univocally defined by the couple (source-address, destination-address) and characterized by several properties derived from the aggregation of packet-based features. The Netflow features we use in our system are listed in Table 6.1.

Table 6.1: Netflow features, with type and examples.

FEATURE	TYPE	VALUES
source-ip	string	147.32.84.193
start-time	timestamp	2011-08-17 15:51:08.499
protocol	string	TCP, UDP
duration	float	0.103, 2.696
direction	string	$\rightarrow, \leftarrow, \leftrightarrow$
total-packets	integer	9, 1
total-bytes	integer	1030, 66, 43

We use Netflow records instead of packet captures because these are frequently logged by network operators and much easier to obtain. Furthermore, they preserve privacy of the communication: in contrast to network packets, Netflows do not contain the content or format of messages. The downside of learning from Netflows is that the learned machines define behaviors on a high abstraction level. In the following sections, we show

that these high level machines are very powerful behavioral models, capable of detecting other infected hosts with very few false positives.

6.3.2. OBTAINING TIMED MESSAGES

Clustering of Netflows basically consists of assigning a numerical code to each flow based on the features in Table 6.1 such that similar Netflows receive the same code. We obtain these using a simple attribute mapping for each feature. Protocol type and direction are assigned a progressive non-negative number for every possible value v, for example, for protocol type, we assign 0 if v = TCP, 1 if v = UDP, 2 if v = ICMP, etc. The source-ip feature is only used to distinguish flows from each other, and the timestamp is used to compute time values. We use percentiles for clustering other numerical features, i.e. duration, total-packets and total-bytes. The ELBOW methods are applied to select the "optimal" number of bins [143]. The experiments show that within-cluster sum of squares (WCSS) has a "break point" at number of cluster-5, i.e. 20th, 40th, 60th, and 80th percentiles. We assign values accordingly: 0 if v is before the 20th percentile, 1 if v is after the 20th and before the 40th, etc. We then compute the event types from Netflows using Algorithm 12, where $M_i : v \rightarrow \mathbb{N}$ denotes the attribute mapping for feature i.

Algorithm 12 Netflow encoding using attribute mappings.

Require: a Netflow $n = \langle a_0, a_1, \cdots, a_k \rangle$ with k features, and an attribute mapping M_i, $i = 0, 1, \cdots, k$ $code \leftarrow 0$ $spaceSize \leftarrow \prod_{i=0}^{k} |M_i|$
for $i \leftarrow 0$ **to** k **do** $code \leftarrow code + M_i(a_i) \times \frac{spaceSize}{|M_i|}$ $spaceSize \leftarrow \frac{spaceSize}{|M_i|}$
end forreturn code

Algorithm 12 uses attribute mappings to encode a Netflow Note that $|M_i|$ denotes the number of values for feature a_i. For example, a simplified scenario where every Netflow has only two features: protocol and total-packets. For the protocol we observe only two possible values, namely TCP and UDP, and the attribute mapping will assign 0 to the former and 1 to the latter. For the total-packet feature let assume we have gathered values: $\{1, 1, 1, 5, 12, 14, 14, 18, 23, 31\}$ and assume we are interested in the 20th and 80th percentiles. We first need to find out the ordinal ranks of such values in the above collection, using the formula: $r(p) = \left\lceil \frac{p}{100} \times N \right\rceil$, where p is the required percentile, and N is the collection size. Therefore $r(20) = \left\lceil \frac{20}{100} \times 10 \right\rceil = 2$ and $r(80) = \left\lceil \frac{80}{100} \times 10 \right\rceil = 8$. Percentiles are the collection values corresponding to the ordinal ranks, thus the 20th percentile is 1 and the 80th percentile is 18, they induce an attribute mapping to $M_{total-packets}(v) = \{0 \text{ if } v \leq 1,\ 1 \text{ if } 1 < v \leq 18,\ 2 \text{ else}\}$. where v is the total-packets attribute value for any given Netflow . Hence the code associated to the instance $\langle$TCP, 14$\rangle$ is: $0 \times \frac{6}{2} + 1 \times \frac{3}{3} = 1$. And the code assigned to the instance $\langle$TCP, 33$\rangle$ is 2. Time values are obtained from Netflow data by calculating the delays between two consecutive events. For each host of interest, we compute the time differences of its consecutive flows in milliseconds. Table 6.2 shows the results of this process applied to five example Netflow records.

Table 6.2: Netflow events from Scenario 9.

prot	dir	time	duration	packet	byte	event
udp	→	0	0.000304	1	68	(1,0)
udp	↔	5	0.000442	5	590	(52,5)
tcp	↔	17	0.000527	3	479	(150,12)
tcp	↔	24	0.120181	6	212	(150,2)
udp	→	22	0.17121	10	7701	(44,5)
...	...	...	...	...	...	...

6.3.3. SLIDING A TIMED WINDOW

As mentioned in Section 6.3, RTI+ learns PDRTAs from sets of timed strings. In this section we address the task of obtaining timed strings from Netflows, achieved in two stages: At first stage we group Netflows by source address (source-ip feature in Table 6.1) because we are interested in modeling per-host behavior. By doing so, we collect a plain sequence of Netflows for each monitored host, which is translated in timed events as showed in the previous section. At second stage we slide a time window of fixed duration over the sequence obtained at stage one. For every window w, we create a timed string by concatenating all events that occur within the duration of w. The window duration used in the experiments is 20 milliseconds, which creates two flows from the data in Table 6.2: $(1,0)(52,5)(150,12)$ and $(52,5)(150,12)(150,2)(44,5)$. Each flow represents a snapshot of 20 milliseconds of timed events produced by a given host, which we call a "Sliding Timed Frame".

6

6.3.4. RECOGNIZING A HOST AS INFECTED

After obtaining a PDRTA A from a malicious host M, we use it to evaluate other hosts C. Intuitively, we compare the expected behavior of a malicious host M, given by the model A with the observed behavior in C. A symptom of C describes how the behavior of C fits the behavior of M. Formally, a symptom is an internal state of A together with the input needed to reach it:

Definition 22. An infection symptom is a couple $\langle q, t\rangle$, where

- $q \in Q_M$ is a state of $\mathscr{A}_M$, PDRTA learned from a given blacklisted entry M; q is the state reached in $\mathscr{A}_M$ after receiving t;
- $t = \langle s, \delta\rangle$, is a timed event with $s \in \Sigma_M$, the set of event types for $\mathscr{A}_M$, and $\delta \in \mathbb{N}$, produced by a given monitored host C;

Every infection symptom is a behavioral fingerprint for a monitored host, given the infection represented by a blacklisted entry. Such fingerprints have a cross-network component (q) obtained from M, and a network-specific component (t), generated from Netflows captured within the NUO. Indeed one might imagine two or more networks to share the same behavioral model M for a specific infection, and still to be able to generate infection symptoms which are specific for each of them. That is why we refer to PDRTA models as *fingerprint generators*.

It is important to underline once more that infection symptoms are generated partially by using a PDRTA learned on Netflows coming from a malicious source (training Netflows data) and partially from data gathered on the monitored network (evaluation Netflows data). To give an example, consider the PDRTA shown in Figure 6.1 and assume to have collected the following frame from the NUO:

$$\{(\text{Q-UDP}, 171)(\text{Q-TCP}, 8)(\text{Q-TCP}, 0)(\text{Q-TCP}, 0)(\text{Q-TCP}, 0)(\text{Q-TCP}, 0)\}$$

The infection symptoms generated are ⟨STATE-5, (Q-UDP, 171)⟩ and ⟨STATE-2, (Q-TCP, 8)⟩, and four instances of ⟨STATE-2, (Q-TCP, 0)⟩.

Once we have learned a PDRTA as a fingerprint for a given malicious host M, we use two different strategies for finding the same infection in a new host C in newly observed data. Both strategies rely on infection symptoms.

Our first strategy, called *error based*, compares the infection symptoms with occurrence counts of the same fingerprints in new data. We thus compare whether a new candidate host C shows the same symptoms as a known malicious host M. Let $Counts_i^M$ and $Counts_i^C$ be counts of symptom i in M and C, respectively. Host C is classified as infected if the absolute error $S = \sum_i |Counts_i^M - Counts_i^C| < \tau$, i.e., if absolute differences between the expected and observed symptom counts fall below a pre-computed threshold. This threshold is obtained using a configuration dataset of known benign hosts; see Section 6.4. The absolute error S can be used as a score function. Using it, we can rank i different hosts C_i according to suspiciousness S. The second strategy called *fingerprint based*, uses this configuration dataset to find distinguishing symptoms that occur when a host is malicious, but never when it is benign. $Counts_i^F$ denotes the sums of all symptom counts in the configuration set.

Host C is then classified as malicious if a symptom i exists, such as $Counts_i^F = 0$, $Counts_i^M > 0$, and $Counts_i^C > 0$.

6.3.5. CIRCUMVENTING PDRTAS DETECTION

At present, BASTA should not be considered as a complete system for the detection of botnets. It has rather been designed as a machine learning engine of a more complex detection device (e.g., random forest module in [141]). Having said that, it is good to clarify that BASTA is a botnet fingerprinting system. This means you need traffic from an infected host known as such, to get a PDRTA representing its behavior and the fingerprint generator. It is also important to point out that thanks to these generators, fingerprints have the desired property of being specific for the network. In particular, to circumvent the detection of BASTA, a bot master should have access to the traffic of the network he intends to attack in order to craft elusive flows and bypass its fingerprints.

6.4. EXPERIMENTS

BASTA is evaluated on a dataset released by Garcia et al [124]. The dataset is organized in 13 different scenarios, each of them containing Netflows from a network infected by

a different type of malware. The scenarios are numbered from 1 to 13, and referred to by their number. The goal of BASTA is identification of other infected hosts knowing at least one, e.g. from a blacklist. Our experiments focus on the four scenarios containing multiple bots running the same malware. Using the same scenario numeration as in [124], Scenario 9 contains a network infected by Neris, a spamming botnet that operates though IRC (Internet Relay Chat) and is capable of performing port scanning and click frauds. Scenario 10 and 11 contain a network infected by Rbot, a botnet capable of leading distributed denial of service attacks (DDoS). The UDP and ICMP protocol are used respectively. Scenario 12 is a network infected by NSIS.ay, a trojan capable of coordinating DDoS attacks.

In [124] several existing botnet detection methods are compared on this dataset. The different methods are trained on samples from Scenarios 3, 4, 5, 7, 10, 11, 12, and 13, and evaluated on Scenarios 1, 2, 6, 8, and 9. The purpose of this setup is to test whether the methods are able to generalize from one botnet to another. In addition, the methods are evaluated on how quickly they can detect new threats. Understandably, the methods perform poorly on this task, sometimes even worse than a baseline that labels all flows as malicious.

For a network administrator it is much more useful to have labels assigned to hosts instead of individual Netflow records because hosts can be investigated and reset to a trusted state. We therefore focus on labeling and ranking hosts.

Our initial goal is to detect new infections of known threats. A network administrator can learn models for such threats using Netflows that connect to known blacklisted hosts. Afterwards, we consider the setting used in [124] of trying to detect infections by unknown threats.

We preprocessed the data, removing null values if present, and accounted for discrepancies in date formats across multiple scenarios. We also removed all Netflows labeled as background as our use-case is a binary problem: discovering whether a given host has or not has been infected by a malware. For each scenario we constructed three disjoint datasets. Table 6.3 summarizes the sets.

- The configuration dataset, containing 30% of the sequences, randomly selected. It used for calculating percentiles in features with numeric domain. It is also used for estimating the selectivity threshold for the error based strategy and for identifying the distinguishing symptoms used in the fingerprint based strategy. It does not contain Netflow sequences from botnet hosts.
- The training dataset, containing all Netflows coming from one of the infected hosts.
- The evaluation dataset, containing the remaining sequences from the scenario: both infected and benign Netflows from all other hosts. The evaluation dataset sometimes is much bigger than the training set. The reason for this is in how we learn: model is learned from just one host while the objective is to detect other infected machines in a big network.

Table 6.3: The number of flows (hosts) of each of the three set per scenario.

Scenario	Configuration Set	Training Set	Evaluation Set	Infected Hosts
9	91386 (185)	29712 (1)	648627 (1077)	10
10	159995 (141)	19889 (1)	465462 (380)	10
11	1004 (32)	138077 (1)	149821 (87)	3
12	6506 (4)	807 (1)	2483 (18)	3

In all the experiments, the attribute mapper has been initialized using the configuration dataset with 20%, 40%, 60%, 80% as percentiles for all scenarios except for Scenario 11, where only the 50% percentile was used in order to avoid overfitting on a small training dataset. Regarding the selectivity threshold used in the error based strategy, it has been estimated by collecting the sum of errors for each host in the configuration set and computing the average error μ along with the standard deviation σ. The threshold has then been set to $\tau = \mu - 2\sigma$.

The following subsections discuss two different types of experiments. Section 6.4.1 shows performances by taking each scenario individually. These experiments illustrate the capabilities of the system in the task of detecting a host infected by a known threat. Section 6.4.2 shows an experiment involving all scenarios together. This experiment aims to assess BASTA performance with threats it does not know anything about, i.e. it cannot rely on a model learned on purpose for such a menace.

6.4.1. PER-SCENARIO EVALUATION

All experiments in this section share the aim of evaluating the system in detecting an already known infection. A known infection is a host known to be infected by prior knowledge, e.g. through a blacklist. In each experiment we compare results with a BIGRAMS baseline.

BIGRAMS are essentially sequences of two consecutive events. For instance, if we consider the stream in Table 6.2, we can get the following: $(1,52), (52,150), (150,150), (150,44)$. This baseline does not use any serialization, i.e. no sliding frames, no time information, just the labels. With BIGRAMS it is possible to estimate the conditional probability of the events. If we have the bigram (E_1, E_2) where E_1 and E_2 are consecutive events, the conditional probability $P(E_2|E_1)$ can be estimated by computing the join probability of $(E1, E2)$ and the marginal probability of $E1$. Performance is presented in terms of true/false positives/-negatives, where a true positive (TP in the tables) means a host correctly classified as malicious.

NOISELESS SOURCES, NOISELESS TARGETS

Table 6.4 refers to a first setting where both source and targets are noiseless. In this context the word "noiseless" is used to express that every Netflow coming from any evaluation machine is legitimate if and only if such a machine is legitimate, and every Netflow coming from any evaluation and infected machine is actually malicious as well. The results on three out of four scenarios are impressive: 100% accuracy for the error based strategy and few false positives for the fingerprint based strategy. Scenarios 11 and 12 show somewhat worse results. One host in Scenario 11 is falsely identified as benign (one false negative) out of the two malicious hosts in the testing data. There is also an

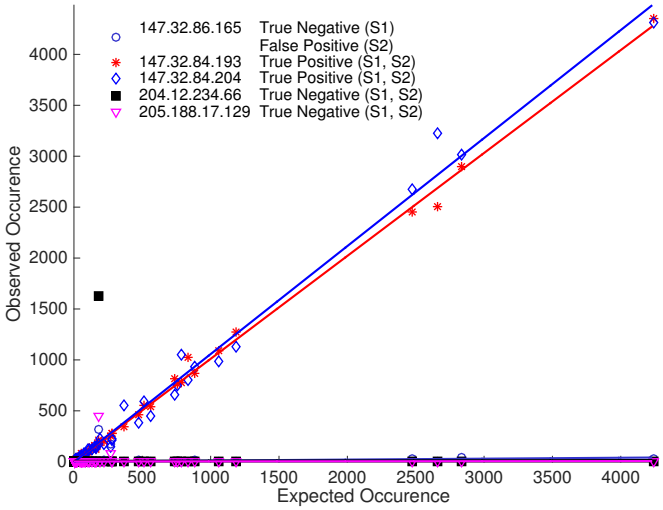


Figure 6.2: Expected and observed frequencies of infection symptoms in Scenario 9. Each $x-y$ pair indicates the expected frequency count versus the observed frequency count of a specific infection fingerprint for one of five given hosts. Regression lines are also drawn for each host. For true negatives, we observe low counts while a fingerprint of an infection expects high counts. The regression line is along the x axis. For true positives, the observed and expected counts match, and the regression line is $y = x$.

improvement in comparison with BIGRAMS on Scenarios 11 and 12, where the baseline is not able to detect any infected host. BIGRAMS seem to work better with the fingerprint based strategy, introducing less false positives and hitting all the malicious machines. Figure 6.2 illustrates the expected and observed frequency distribution in each testing hosts. $S1$ and $S2$ are the error based and fingerprint based strategies, respectively. Two hosts are detected correctly as infected (TP) by $S1$ as their regression plot is very close to $y = x$. The other three hosts are correctly detected as safe (TN). Interestingly, one host is incorrectly detected as infected by $S2$. Although this host is behaviorally very different from what the PDRTA model expect, it shows malicious behavior that never occurred in the configuration dataset, in this case the symptom $\langle s = 1, t = \langle 3, 4 \rangle \rangle$ (see Figure 6.1).

Results in Table 6.5 are about a different setting where the sources for each scenario are still noiseless, but the infected evaluation targets are not. This is a more realistic situation where a fingerprint generator, namely the PDRTA, of an infection has been provided by some partners (e.g. security companies) able to run the infection in a safe environment. However, the target infected machines present mixed traffic made of both Netflows labeled as malicious and legitimate. We have devised this setting coupling each malicious host with the most verbose (in terms of number of NetFlows), legitimate, and available one. Then we have merged Netflows of the coupled host so creating new targets with mixed behavior. Even in this case we observe no false positive with the error based strategy, and an improvement in Scenario 9 on the baseline. BIGRAMS with fingerprint based

Table 6.4: Top: Error based strategy performances on noiseless data. Bottom: Fingerprint based strategy performances. The table reports hosts correctly and incorrectly identified in absolute numbers.

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	9	1068	0	0	9	1068	0	0
10	9	371	0	0	9	371	0	0
11	1	85	0	1	0	85	0	2
12	2	16	0	0	0	16	0	2

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	9	1038	30	0	9	1066	2	0
10	9	370	1	0	9	369	2	0
11	1	80	5	1	1	83	0	2
12	2	4	12	0	2	15	1	0

Table 6.5: Top: Error based strategy performances with noiseless sources and noised targets. Bottom: Fingerprint based strategy performances. The table reports hosts correctly and incorrectly identified in absolute numbers.

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	4	1244	0	5	1	1244	0	8
10	0	503	0	9	0	503	0	9
11	1	115	0	1	1	115	0	1
12	0	18	0	2	0	18	0	2

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	9	1219	25	0	9	1244	0	0
10	9	503	0	0	9	503	0	0
11	1	111	4	1	1	114	1	1
12	2	6	12	0	2	17	1	0

strategy show better performance than the alternative system, confirming the trend of the previous setting.

NOISED SOURCES, NOISED TARGETS

The last experiment, whose results are reported in Table 6.6, combines noised sources (obtained with the same previously described coupling procedure) and noised targets. Even with a comprehensible drop in performance compared with already described experiments, the error based strategy still shows better performance on all scenarios but 12. In this case both systems are unable to detect any infected machine, but only the alternative system returning a false positive. Hence, even with the fingerprint based strategy,the trend of the previous setting is unconfirmed and BASTA achieves better results on Scenario 9 and 11 making the comparison with the baseline more uncertain.

INCLUDING BACKGROUND TRAFFIC

In Section 6.4 we explained why, as a pre-processing stage, we decided to filter out the background traffic from the dataset at our disposal. On one hand we don't know the actual label of every background flow, i.e. whether it is malicious or not, posing more than

Table 6.6: Top: Error based strategy performances with noised sources and noised targets. Bottom: Fingerprint based strategy performances. The table reports hosts correctly and incorrectly identified in absolute numbers.

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	2	1244	0	7	1	1244	0	8
10	2	503	0	7	2	503	0	7
11	2	101	14	0	1	98	17	1
12	0	17	1	2	0	18	0	2

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	9	1228	16	0	1	1244	0	8
10	9	503	0	0	9	503	0	0
11	1	98	17	1	1	89	26	1
12	2	5	13	0	2	17	1	0

Table 6.7: Top: Error based strategy performances on noiseless data including background flows. Bottom: Fingerprint based strategy performances. The table reports hosts correctly and incorrectly identified in absolute numbers.

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	9	237381	0	0	9	237381	0	0
10	9	129596	0	0	4	129596	0	5
11	1	20609	0	1	0	20609	0	2
12	0	41687	0	2	0	41687	0	2

	BASTA				BIGRAMS			
	TP	TN	FP	FN	TP	TN	FP	FN
9	9	237340	41	0	9	237320	61	0
10	9	129594	2	0	9	129592	4	0
11	1	20607	2	1	0	20607	2	2
12	2	41667	20	0	2	41668	19	0

few questions (e.g. how to evaluate a legitimate host which includes some background flows). On the other hand, by including background traffic, we reproduce a more realistic setting in which we are interested to test BASTA performance. Finally we decided to plan additional experiments on single scenarios, with noiseless data, including background traffic. We treated background flows in a neutral way: if a legitimate host contains background flows it remains legitimate, and hosts containing background flows only are considered not malicious. Results of our experiments are showed in Table 6.7, where the trends of the same experiment without background traffic (Table 6.4) have been confirmed.

6.4.2. MULTI-SCENARIO EVALUATION

We also test our system on the setup described in the paper [124] published with the Netflow dataset. The training set consists of samples from scenarios 3, 4, 5, 7, 10, 11, 12, 13, and the evaluation sets contains hosts from scenarios 1, 2, 6, 8, 9. It is important to mention that we selected training and evaluation scenarios as in the corresponding experiment in [124]. The scenarios in the training set contain different botnet families

Table 6.8: Performance on the setup in [124] with error based strategy (top) and fingerprint based strategy (bottom). Scenarios 1,2,6, and 8 only contain a single malicious host, Scenario 9 contains 10 malicious hosts. Overall, the evaluation set contains 3087 hosts.

Scenario	1	2	6	8	9
TP	1	1	1	1	10
FP	3	73	55	84	109
TP	1	1	1	1	10
FP	4	26	12	52	72

than the evaluation set and can therefore exhibit very different behavior. We learned PDRTAs for all malicious hosts in the training samples, and tested whether any of them mark any of the hosts in the testing scenarios as malicious. In spite of our method focusing on behavioral detection, the fingerprint based method produces encouraging initial results, see bottom half of Table 6.8. Overall our system detects all the infected hosts and produces 166 false positives out of 3072 benign hosts. The error based strategy is able to detect all 14 infected hosts, but the performance are deteriorated by an almost doubled amount of false positives (i.e. 324, see Table 6.8, top half).

7

Detecting Intrusions with Probabilistic Deterministic Automata

In this chapter we propose a detection technique which leverages un-timed automata called Probabilistic Deterministic Automata. The chapter concludes the third part of the thesis, and its main aim is to provide a full description of a behavioural detection system, representing the "summa"[1] of what has been discussed along this thesis.
The chapter starts with the data abstraction process, addressed by Section 7.2, and continues introducing the Probabilistic Deterministic Automata (PDAs) in Section 7.3. Eventually, it discusses the problem of automatically infer PDAs from network data - Section 7.4. Section 7.5 is where the problem of how to efficiently use PDAs for detection is addressed. Section 7.6 describes the type of data has been used in the experiments of Section 7.7, which also compares the proposed system with a widely used alternative. Section 7.8 highlights one benefit of using Automata for detection: interpretability. By interpreting PDAs, we show it is possible to obtain actionable insights to enforce the cyber-security of a network. Section 7.9 address the limitations of the proposed detection method.

7.1. INTRODUCTION

More than 250 Distributed Denial of Service (DDoS) attacks have been recorded per day in the third quarter of 2017 [144]. DDoS attacks are a mean to pursue political aims, and blackmail organisations [145]. Botnets are the primary weapon for executing DDoS campaigns, consisting of a network of compromised hosts remotely controlled by a human botmaster. Detection forms the first line of defence against botnets. Network analysers are detection tools that base their detection on a fundamental property of any host included in a botnet: the need of communicating with the botmaster or with other infected hosts [146]. A network analyser's primary goal consists of monitoring the communications and raising alarms in case of suspicious activity [127, 147]. Due to their robustness, behaviour-based intrusion detection systems are among the most popular network analysers [135–137]. Indeed, a botnet may change its binary code at run-time using polymorphic capabilities [134], or automatically select which exploit to leverage [148]. But it is unlikely to modify its means of communication as a change of this magnitude would require coordination between all hosts it communicates with and essentially produce a new botnet.

In this work, we present a new behaviour-based intrusion detection method. Our method describes a host's observed behaviours using a particular type of state machines called Probabilistic Deterministic Automata (PDAs). State machines are a standard tool for describing software specifications [56], and botnets infections are a form of software running on every compromised host. This aspect makes state machines an ideal tool for their analysis, see [138] for an excellent example. In this work, we automatically learn state machines from network flow data. For privacy reasons, we do not inspect the packet content. Since the behaviour of hosts and botnets is known to change over time, we learn these models from small chunks of flow data consisting of 1000 flow sequences. Those are sufficient data for a state-of-the-art PDA learning algorithm to return insightful models. We use a library of such models to detect infected hosts displaying behaviour similar to that of a known infection. These PDAs have been learned from both benign

[1] Medieval Latin word meaning summary, recap.

and malicious network flow data. The benign data comes from hosts in the network under observation. The malicious data comes from hosts with known infections and the PDAs describing these can be learned remotely. With those abstract descriptions of both malicious and normal behaviours, we show how to generate profiles that are specific to the network under observation. All network-specific profiles are compared to a profile constructed from network flow data from an unknown host using our distance measure. Our method searches for the closest known profile: if the nearest profile is an infection profile, the host is classified as infected.

Behavior-based intrusion detection systems have also been used as anomaly detection systems [149–152]. Anomaly detection is the process of modelling a host's normal behaviour to find anomalies and misuses, for instance by raising an alarm when a new data profile shows large distances to all known profiles. Since this does not require known malicious data as input, it could be used to find new unknown threats. Anomaly detection, however, is much harder than intrusion detection where examples of malicious activities are known, and they consequently suffer from high false alarm rates. In this chapter, we assume the availability of malicious examples. This is not unrealistic: malware research centres have quick access to botnet data. Our method allows them to automatically learn models from this data to detect these threats in different computer networks. Since the models capture a botnet's behaviour over time, their detection is much harder to evade than the current rule-based approach used in industry. Also, since we also make use of the local hosts' network flow data during detection, this evasion becomes even harder. The botnet essentially has to behave similarly to a host in the local network to evade detection. Since it is also known that different malware show similar behaviours [153], our method can be used to find and detect such similarities.

7.2. DATA ABSTRACTION

Our method aims to model host behaviours with Probabilistic Deterministic Automata, a particular type of state machine. Almost all the state-of-the-art automata learning algorithms require a symbolic abstraction of the original data. Our proposed method is based on learning PDAs and works in the same way, by converting network flows into a collection of strings made of discrete symbols.

The input data for our algorithm are network flows. A network flow, also commonly referred as NetFlow [63–65] is an aggregation of packets that share a common key. The key is usually the 5-tuple comprised of the source IP address, source port, destination IP address, destination port, and protocol. All the packets that share this key in a specific time frame, are considered part of the same flow. A flow summarises specific attributes of the packets as the duration, the size in bytes, and the number of packets. Network flows do not consider the content of the packets. Hence they are encryption unaware, more privacy-preserving, and easier to obtain and share than packet captures. From these flows attributes, our algorithm only uses the following four: *protocol*, *duration*, *amount of packets*, *bytes*. This selection of attributes is based on two desired properties of our algorithm. First, the resulting symbols should be as easy to interpret for a human as possible. Easiness of interpretation, in this case, means that each symbol should represent a combination of few attributes and, more important, the relation among them

should be easy to spot. We will provide real examples of those concepts in the next subsection. Second, the model should capture the behaviour of the host independently of the ports, IP addresses, and services used. Although these attributes maybe useful for rule-based intrusion detection systems, we believe their values to be too fine-grained to show interesting sequential behavior.

In the following, we describe the design of the alphabet from the flows and the creation of the strings of symbols to be used for learning our behavioural models. Since our method is host centric, there will be a different alphabet, a different sample of strings, and consequently a different model for each considered host.

7.2.1. ASSESSING THE MESSAGE TYPES

An alphabet is a finite set of symbols. In our work, each symbol represents an event in the network that may cause a change in a host's behaviour. To create a symbol, we start analyzing each flow in the network. For each flow, four attributes are extracted: protocol, duration, amount of packets and amount of bytes. The values of these four attributes are discretized for each attribute type. Once the individual attributes are discretized, each combination of discrete values is assigned a different symbol. Each flow is thus represented using a single symbol.

The *protocol* attribute is categorical since it has a predefined and limited set of values, such as TCP, UDP or ARP. Hence it is already a feature with discrete values. The discretization process simply assigns a different code to each observed category. Since the abstraction is used to learn state machines, all categories should occur sufficiently frequent. An additional category is therefore added to cover all infrequent values. This category is also assigned to protocol values that did not occur when learning the model, but do occur when detecting intrusions.

Total Packets, *Total Bytes*, and *Duration* are numerical attributes. Their discretization process consists in splitting the feature interval in two by using the median (50th percentile). As an example of discretization of the Total Packets feature, let's assume that the Total Packets values for the network flows and for a given host are the following: 4×5 times, 16×9 times, and 43×14 times. The median of those values is 29.5, which allows the identification of two partitions: $I_{low} = [0, 29.5]$ and $I_{high} =]29.5, \infty[$. If the number of flows on either interval is deemed insufficient for learning a state machine, the two intervals are combined into one. In such cases, there will be a single interval spanning trough the whole data domain. This case corresponds to an attribute with always the same symbolic abstraction regardless of its actual values.

When all attributes have been discretized, we obtain an alphabet by assigning a unique symbol to each possible combination of discrete attribute values. For the Protocol and Total Packets attributes in the above examples, this results in 6 symbols since there are 3 possible Protocol values and 2 possible Total Packet values. Given this discretization, let us consider a flow with Protocol=UDP and amount of Total Packets=66. This flow will be assigned the symbol corresponding to a UDP flow, with Total Packets in the range $]29.5, \infty[$. This performed discretization is model dependent, meaning that for different models (hosts), different symbols will be used depending on their occurrence fre-

quencies and value distributions. It is fully automated and requires no human intervention.

7.2.2. CRAFTING SUBSEQUENCES

Once the alphabet is created, we transform the flows produced by a host into a sequence of symbols. This sequence is further decomposed into smaller groups of consecutive symbols, i.e., *strings*. The algorithm that generates these strings is a fixed-size sliding window that creates a new string every time it moves. For example, using a sliding window of size 3, the symbolic series $S = [1,1,2,1,5,2]$ is decomposed into the following 4 strings: $s_0 = [1,1,2]$, $s_1 = [1,2,1]$, $s_2 = [2,1,5]$, and $s_3 = [1,5,2]$.

The size of the time window was chosen recurring to the autocorrelation plots [154]. Autocorrelation plots are used for checking the randomness of a time series by correlating it with itself with varying time lags. If a sequence of symbols is random, then the autocorrelation plot will be close to zero at each time lag. However, if the sequence presents some regularities such as cyclic behavior or recurring patterns, then we may expect at least a time lag in the plot whose autocorrelation score will be significantly non-zero. We set the window size to the smallest time lag with no significant autocorrelation. The rationale in using this technique relies in the goal of using our models for describing recurrent patterns (symptoms) in a host's observed behavior. If we would include subsequences with zero autocorrelation, the state machine learning algorithm will try to integrate these with the model. This could still be used to find new patterns, but there is a high risk of introducing noise to the patterns it already discovered. We therefore set the sliding window to be the largest length without this property. Figure 7.1 shows the autocorrelation plot for the symbols generated from 1,000 flows coming from a virtual host infected with Neris malware. There is no significant autocorrelation at lag 19 (the solid gray line represents the significance threshold). There is also no significant autocorrelation at lags 24 and 28, but lag 19 is the smallest. This gives $19 - 1 = 18$ as sliding window size. Over all hosts in the dataset, the sliding window sizes range from X to Y, with Z being the most frequent size. Note that every host gets assigned its own window size. Like the alphabet creation, this process is fully automated.

7.3. PROBABILISTIC DETERMINISTIC AUTOMATA

The models we use for describing host behaviours are called Probabilistic Deterministic Automata (PDAs). PDAs are the probabilistic counterpart of Deterministic Finite State Automata. As state machines, PDAs can change from one state to another in response to some external input represented by a symbol. PDAs also describe distributions over strings, having a Markov property over the symbol distribution.

Definition 23. A PDA is a 5-tuple $\langle Q, \Sigma, q_0, \Delta, \mathscr{E} \rangle$, where

- Q is a finite set of states.
- Σ is a finite set of symbols.
- $q_0 \in Q$ is the start state.

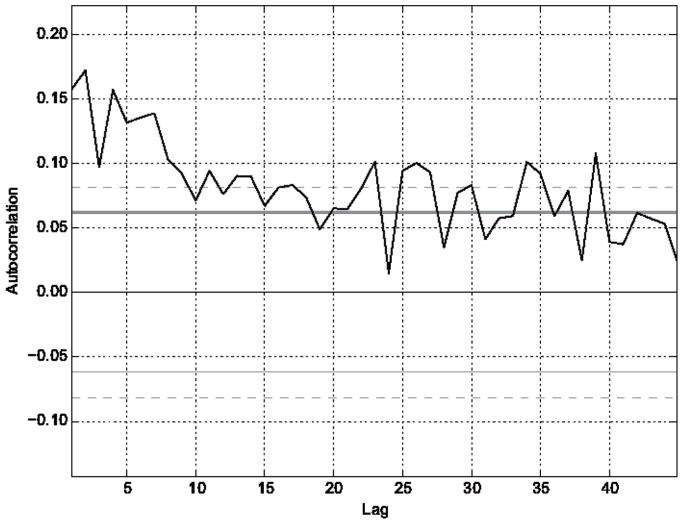


Figure 7.1: Autocorrelation plot for the symbols obtained from 1,000 flows from a host infected with Neris malware.

- Δ is a finite set of transitions.
- $\mathscr{E}$ is a symbol probability distribution such that $\mathscr{E} = (Q, \Sigma) \rightarrow [0,1]$, returning the probability of generating a symbol in a given state.

A transition $\delta \in \Delta$ in a PDA is represented by the triple $\langle q, q', a \rangle$, where $q, q' \in Q$ are the source and target states, and $a \in \Sigma$ is a symbol. The 4-tuple $\langle Q, \Sigma, q_0, \Delta \rangle$ is called *structure* of the PDA. PDAs are deterministic, meaning that there are no ambiguities on which transition to follow. On every state exists only one transition for each symbol in the alphabet.

Definition 24. A run on a PDA $\mathscr{A} = \langle Q, \Sigma, q_0, \Delta, \mathscr{E} \rangle$ of the string $s = s_1, s_2, \ldots, s_M$ is a finite sequence

$$q_0 \xrightarrow{s_1} q_1 \xrightarrow{s_2} q_2, \ldots, q_{M-1} \xrightarrow{s_M} q_M$$

such that $\langle q_{i-1}, q'_i, s_i \rangle \in \Delta$ for for $i = 1, 2, \ldots, M$.

Figure 7.2 and shows a PDA $\mathscr{A}$ inferred from a host infected by Rbot malware. In this machine, string $s = [\text{FL}, \text{FL}, \text{FL}, \text{SL}]$, traverses the run $\rho = [0, 2, 3, 3, 1]$ over the state of $\mathscr{A}$. It is also possible to compute the likelihood of s being generated by $\mathscr{A}$ by applying a chain-like rule: $L(s|\mathscr{A}) = \mathscr{E}(0, \text{FL}) \times \mathscr{E}(2, \text{FL}) \times \mathscr{E}(3, \text{FL}) \times \mathscr{E}(3, \text{SL}) = 0.48 \times 0.59 \times 0.72 \times 0.27 = 0.055$.

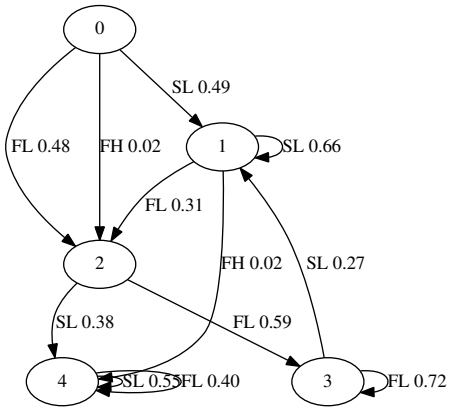


Figure 7.2: Probabilistic Deterministic Automaton (PDA) identified from network flows produced by a host infected by Rbot. State 0 is the start state. Along with each transition is the symbol probability. The transitions with a low probability have been omitted to improve readability. The symbols stands for a Slow and Light (SL) flow, a Fast and Heavy (FH) flow, and a Fast and Light (FL). A Short duration means less than 1.201 milliseconds. A light flow means less than 2 packets and fewer than 124 bytes. There are two most likely paths within the automaton: a sequence of SL (state 0, then state 1 ad libitum), and a sequence of FL (state 0, then state 2, then state 3 ad libitum).

7.4. PDAs Inference

The structure of a PDA, together with the distributions, can be constructed fully automatically from observed sequence data. For this, we use an untimed version of the RTI+ algorithm proposed in [108] for learning probabilistic deterministic real-time automata, which we call the Probabilistic Deterministic Automaton Identifier (PI). Given a sample of strings S, the goal of PI is to find that PDA which better describes the process that generated S. PI is an evidence-driven state merging algorithm (EDSM) that implements the so-called red-blue framework. At any time during its execution, PI keeps a set of red states representing the already discovered parts of the automaton. Red states are finalized, meaning once a state is coloured red it will never get updated anymore. Furthermore, PI keeps a set of blue states representing the part of the automaton has already been discovered, but not finalized. A blue state may evolve, by promotion, into a new red state, or get merged with an existing red state. The remaining uncoloured states stand for the part of the automaton which has not yet been discovered.

The initial step in every EDSM algorithm is the creation of a prefix tree. A prefix tree is a first guess of the automaton and contains a branch for each different sequence within S. Note that S can be seen as a multiset - i.e. it may contain multiple occurrences of a given string - and PI keeps track of how many times a given transition is traversed by the strings in S. Figure 7.3 shows an instance of a prefix tree created from the provided sample.

As a second step, PI starts to merge the states in a red-blue framework, see [61] for a more detailed discussion of this framework and state merging algorithms. Figure 7.4 shows the consequences of merging two states in a prefix tree. By merging states, the model gains

in abstraction power, i.e., it will possibly express more behaviour than what explicitly observed in the sample. As an example, The automaton in Figure 7.4 can generate the sequence aaa which was not possible to produce in the prefix tree. After merging states q and q', the counts are updated, transitions landing in q' are redirected to q, transitions leaving q' are attached to q, q' is removed from the automaton, and the transition counts are updated. Since PDAs are deterministic by definition, if q now contains two transitions with the same transition symbol, the states reached by these transitions have to be merged as well. This deterministic property is important because identifying a non-deterministic automata is considered harder than identifying a deterministic one, see [109] for more information.

Merging states results in a smaller and simpler model, as well as a smaller likelihood. The PI algorithm makes a trade-off between this decrease in likelihood and the increase in model simplicity. Essentially, a merge between q and q' is regarded as *consistent* if the future behavior after reaching q is similar to the behavior after reaching q'. As is done by RTI+ [108], this similarity is tested with a likelihood-ratio test which essentially checks the Markov property. It verifies whether the future behaviour is independent of being in state q or q'. When these two futures are significantly different, the merge is considered *inconsistent* and it is not performed.

Algorithm 13 Learning PDAs from samples of strings: PI algorithm

Require: A multi-set of strings S
Ensure: The result is a PDA $\mathscr{A}$
  Construct a prefix tree $\mathscr{A}$ from S, color the start state q_0 of $\mathscr{A}$ red
  while $\mathscr{A}$ contains non-red states **do**
    Color blue all non-red target states of transitions with red source states
    Let $\delta = \langle q_r, q_b, s \rangle$ be the most visited transition from a red to a blue state
    Evaluate all possible merges of q_b with red states
    if the highest merge p-value is greater than 0.05 **then** perform this merge **else** color q_b red

The main loop of the PI algorithm is shown in Listing 13. At each iteration, the algorithm selects the most visited transition which connects a red state to a blue state and considers all possible merges between the blue state and any known red state. If there exists a merge resulting in a p-value over 0.05, the algorithm performs the merge with the highest p-value otherwise it promotes the blue state to red, and all its children to blue. The rationale for choosing the most visited transition is that any decision taken in the two states at both sides are based on the most significant amount of data. Hence the confidence in the conclusion is maximised. The PI algorithm runs in polynomial time in the size of the sample since it is upper bounded by RTI+, which runs in polynomial time in S [108].

7.5. PDAs FOR DETECTION

Once we have learned PDAs from network flow traces, we use them for detecting symptoms of infections. Given a PDA $\mathscr{A} = \langle Q, \Sigma, q_0, \Delta, \mathscr{E} \rangle$, a *symptom* is the couple (q, α) com-

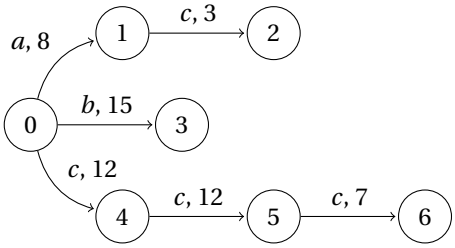


Figure 7.3: Prefix tree created from the sample $S = \{a\times5, ac\times3, b\times15, cc\times5, ccc\times7\}$. The first number represents the symbol triggering a transition. The second number represents its occurence frequency in S.

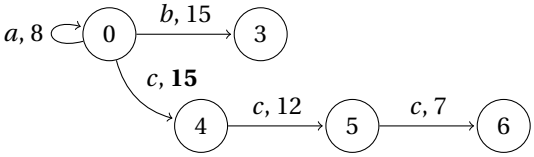


Figure 7.4: A PDA after merging the states 0 and 1 in the prefix tree of Figure 7.3. The size of the automaton is decreased by two states to ensure a deterministic model, and counts (from state 0 to 4) are updated.

posed by a state $q \in Q$, and a symbol $\alpha \in \Sigma$. In a symptom, q is the state of $\mathscr{A}$ reached after producing α. It is possible to generate symptoms for $\mathscr{A}$ given a sample of strings S. For each string $s = s_1, s_2, \ldots, s_M \in S$, and for each symbol s_i in s, with $i = 1, 2, \ldots, M$, collect the state $q_i \in Q$ such that $(q_{i-1}, q_i, s_i) \in \Delta$. This means that the state has been reached in $\mathscr{A}$ after producing the symbol s_i. That state paired with the symbol form the symptom (q_i, s_i). Given a string in the alphabet of $\mathscr{A}$, it is possible to generate as many symptoms as the length of s by pairing its symbols with the states composing a run of s on $\mathscr{A}$. The first column of Table 7.1 shows some symptoms generated by the PDA in Figure 7.2.

Given a PDA $\mathscr{A}$, the set of symptoms it may produce is fixed, and is called the symptom space of $\mathscr{A}$. This is defined as $D_{\mathscr{A}} = \{(q, \alpha) \mid q \in Q \text{ and } \alpha \in \Sigma\}$. A profile, then, can be defined as $\mathscr{P}_{\mathscr{A},S} : D_{\mathscr{A}} \to [0, 1]$, given a sample of strings S. The profile is a probability distribution on the symptom space estimated by maximum likelihood:

$$\mathscr{P}_{\mathscr{A},S} : (q, \alpha) \to \frac{c_S(q, \alpha)}{\sum_{(q', \alpha') \in D_{\mathscr{A}}} c_S(q', \alpha')}$$

With $c_S(q, \alpha)$ denoting the number of times symptom (q, α) has been generated by $\mathscr{A}$ on sample S. Table 7.1 shows the profiles for a host infected by Rbot malware. It only reports symptoms with a not null estimated probability. From the table, it is possible to understand what are the most frequent paths in the PDA. The most likely symptom within the profile is (1, SL) which involves the transition from state 0 to 1 and the self-loop in this state. Symptoms (2, FL) and (3, FL) follow, which corresponds to the path from 0 to 3 passing through 2 and include the self-loop in state 3. It thus becomes possible to understand what are the most frequent recurring patterns in a host's observed data. In this case, we have two dominant patterns: a first one consisting of sequences of slow

and lightweight flows, and a second one composed of sequences of fast and lightweight flows.

Table 7.1: Profile for the PDA in Figure 7.2 given the sample used to learn it. State −1 refers to a *sink* state, a state reached by all rarely occurring transitions.

Symptom	$\mathscr{P}_{\mathscr{A},S}$(Symptom)
(-1, SL)	0.013
(-1, FL)	0.007
(1, SL)	0.341
(-1, FH)	0.011
(2, FL)	0.205
(4, SL)	0.133
(3, FL)	0.225
(2, FH)	0.005
(4, FL)	0.045
(4, FH)	0.006

To detect similar behaiour, we use a *profile distance measure*. Let us assume to have learned a PDA $\mathscr{A}$, learned from a sample S, and let us assume to have collected a sample S' not necessarily produced by the same host. Running $\mathscr{A}$ over these samples create two profiles $\mathscr{P}_{\mathscr{A},S}$ and $\mathscr{P}_{\mathscr{A},S'}$. The profile distance measure $d(\mathscr{P}_{\mathscr{A},S}, \mathscr{P}_{\mathscr{A},S'})$ is defined as the chi-square distance between the two profiles:

$$d(\mathscr{P}_{\mathscr{A},S}, \mathscr{P}_{\mathscr{A},S'}) = \frac{1}{2} \sum_{x \in D_{\mathscr{A}}} \frac{(\mathscr{P}_{\mathscr{A},S}(x) - \mathscr{P}_{\mathscr{A},S'}(x))^2}{\mathscr{P}_{\mathscr{A},S}(x) + \mathscr{P}_{\mathscr{A},S'}(x)}$$

Profiles and the profile distance are the main building block of our detection algorithm, which is divided in two parts: the training part, and the testing part.

The training part of our framework is described in Algorithm 14. It requires labeled network flows from N hosts, i.e., we know which hosts are infected and which are not. We abstract the symbols from the flows, learn a state machine for each training host, and generate a profile for each training host. After the training profiles are created, we test our detection method. Algorithm 15 describes the testing part, where each host in the testing dataset is predicted to be infected or normal. The testing part also includes abstracting the data from the test flows and creating a profile for each of them. Note that profiles are generated very fast because the state machines are deterministic. Then, for each PDA, the method computes the distance between the testing profile and its training profile. The label of the best match is assigned to the test flows.

7.6. DATASET

CTU-13 is a large dataset created at the Czech Technical University in Prague that is composed of 13 different captures [124]. Each scenario has malicious traffic, normal traffic, and background traffic. The malicious traffic was infected by hand-picked malware executables, the normal traffic came from known verified users, and the background traffic

Algorithm 14 Detecting infected hosts with PDAs: training

Require: A collection of network flow samples $\mathscr{S}_0, \mathscr{S}_1, \ldots, \mathscr{S}_N$ produced by hosts $1, 2, \ldots N$
  for each host $h = 1, 2, \ldots, N$ **do**
    Abstract $\mathscr{S}_h$ in a sample of strings S_h
    Learn a PDA $\mathscr{A}_h$ from S_h and store it
    Generate profile $\mathscr{P}_{\mathscr{A}_h, S_h}$ and store it
  end for

Algorithm 15 Detecting infected hosts with PDAs: testing

Require: PDAs and profiles for each host in the training sample, a sample $\mathscr{S}$ of flows produced by a candidate host
Ensure: a verdict in terms of infected (M) or normal (N)
  Set the closest candidate c to the first training host
  for each host $h = 2, 3, \ldots, N$ in the training sample **do**
    Abstract $\mathscr{S}$ in a sample S of strings in the alphabet of $\mathscr{A}_h$
    Generate profiles $\mathscr{P}_{A_h, S}$ and $\mathscr{P}_{A_c, S}$
    if $d(\mathscr{P}_{A_h, S}, \mathscr{P}_{A_h, S_h}) < d(\mathscr{P}_{A_c, S}, \mathscr{P}_{A_c, S_c})$ **then**
      Set the clostest host c to h
    end if
  end for
  if c is infected **then return** M **else return** N

is everything that is unknown. On each scenario, there are one or more computers infected by the same botnet and at least three normal computers. Table 1 in appendix 1 shows the number flows per type of label and IP address on each of the scenarios. On the datasets, there are several more hosts that are marked as normal because they are servers in the network. However the number of flows they produce per dataset can be as low as 6, they are therefore not included as part of the dataset. Their IP addresses are 147.32.87.36 (CVUT-WebServer), 147.32.80.9 (CVUT-DNS-Server), 147.32.87.11 (MatLab-Server).

CTU-206-1 is a 28 days-long execution of a Dridex family malware [2]. The dataset is available in [155]. This version of the Dridex malware uses HTTPs connections in exotic ports such as 4413, 44343 and 8443 to connect to its Command and Control (C&C) servers. The total list of C&C servers is:

- 195.88.209.221:4413
- 178.32.255.130:44343
- 91.219.28.55:443
- 217.197.39.1:8443

[2] MD5 0243c9bb903d6f89d7eeadae882cf591.

Dridex is a banking trojan designed to steal bank credentials and personal information of the user to access bank accounts. Dridex steals personal information from HTML injections, mainly from Europe users. All the malware behaviours in the CTU-260-1 capture are the connections to the C&C servers. There are no DNS connections nor any other type of attack, probably because there was no user accessing bank accounts.

CTU-261-1 is a 63 days-long execution of novel banking trojan called TrickBot[3]. The complete dataset is available in [156]. The behavior of this malware is quite complex and it is an excellent challenge for this research. In summary, the malware did the following actions:

- Finds public IP of the infected host.
- Connects to several C&C servers on port 443/tcp.
- Connects to two C&C servers on port 447/tcp.
- Checks if the infected host is listed as a SPAM sender.
- Connects to a C&C server on pot 2048/tcp.
- Attempts to send SPAM to port 465.
- Attempts to send SPAM to port 587.
- Attempts to send SPAM to port 25.
- Continues the C&C connections on port 443/tcp and 447/tcp. Including connections to new C&C IP addresses.

The TrickBot malware had several different behaviors, including downloading binary files, trying to send SPAM and a very active HTTPS-based Command and Control channel.

7.7. EXPERIMENTS

The process to evaluate an algorithm is one of the most important parts of research in botnet detection. It is during this part that decisions are taken regarding the size of the datasets, the specific ways to split flows and the way to measure the performance of the algorithm. This section describes how our dataset is processed, how the training is performed, how the cross-validation is done and the results obtained from the testing dataset. Finally, we present a comparison of our method with another widely used malware detection method on the same dataset.

7.7.1. VIRTUAL HOSTS

We use the CTU-13 dataset, complemented with new malware capture experiments. The complete dataset description is addressed in Section 7.6. Each host in our dataset is either infected with malware or normal. Although we use flows for creating the PDAs, the

[3] MD5 bb9e0b23fc6cba27ba670547b7890273.

focus of our work is to detect infected hosts instead of suspicious flows. That is an important point because there are far fewer hosts in a dataset than flows or packets. In our data, the number of known hosts is less than 100 hosts. When we evaluated our system on such a small number of candidates, we got 100% accuracy. To solve this situation, and given that each host in the dataset generated a lot of flows, we decided to separate the flows of each host in *chunks* of flows, that we call *virtual hosts*. Each *virtual host* is a consecutive group of flows generated by the same host. Since the behaviour of a host changes over time, our technique can be used to detect the different network behaviours of the hosts. Using this technique, we (1) obtain enough virtual hosts to train and test the method with reliability, and (2) we can apply our method to fewer flows, making it more suitable to be implemented in a real network. The size of the *virtual hosts* was set to be 1000 flows following advice from experts in grammatical inference. Learning PDAs from less than 1000 sequences is considered unreliable. We choose the smallest number of flows that gives reliable results. In our dataset, this gave us chunks ranging from 25 milliseconds to 1 day and 23 hours, with an average of 21 minutes.

7.7.2. WHOLE DATASET EVALUATION

In a first experiment, we separate the complete dataset in training and cross-validation/testing sets with the aim of finding out the general performance of the proposed method. We allow any malware model to detect any other malware virtual host, and any normal model to discover any normal virtual host. We first separate all the dataset in hosts, each host is divided into virtual hosts, and then all the virtual hosts are randomly partitioned into five folds. When creating the partitions, we imposed to have an even proportion of both infected and normal virtual hosts in each of them. That choice led to five folds each one including approximatively 85% normal hosts and 15% infected hosts. Table 7.2 shows the distribution of the virtual hosts in each fold. It is important to remark that all the virtual hosts with less than 1000 flows were discarded since they were not providing enough information to create meaningful PDAs. Table 7.3 summarizes the results of the detection after averaging the 5-folds. TP stands for true positives, TN stands for true negatives, FP stands for false positives, FN stands for false positives, Acc stands for accuracy, FPR stands for false positive rate. Using a 64-bits computer with 8GB of RAM memory and without any optimization, the training part took in average 49 minutes per fold. Classifying a virtual host (comparing it to he entire library of PDAs), took on average 17 seconds.

Table 7.2: Partitioning of 8,363 virtual hosts in five randomly generated folds (F). Each of them preserves the proportion of infected and normal virtual hosts.

	TOTAL	Infected	Normal
F1	**1,674**	227	1,447
F2	**1,674**	227	1,447
F3	**1,674**	227	1,447
F4	**1,674**	227	1,447
F5	**1,667**	222	1,445
TOTAL	**8,363**	**1,130**	**7,233**

Table 7.3: 5-fold cross validation results. Discrete quantities (TP, TN, FP, FN) outcomes, and measurements (Acc, FPR, F1-score) outcomes of each validation round are per-fold averaged.

TP	TN	FP	FN	Acc	FPR	F1-score
223	1,441	5	2	0.995	0.004	0.983

7.7.3. DETECTING A KNOWN THREAT

The goal of this experiment is to assess the algorithm for detecting hosts infected by a known malware. That is a malware that was previously detected and whose data is included in the training dataset. Since we must guarantee that the virtual hosts of some malware are used for both training and testing, we redo the separation of the dataset differently than the previous experiment. The first step is to separate all hosts in the dataset into virtual hosts, but remembering to which real host they belong. Then, we take all the virtual hosts on each host, and we randomly separate 80% of them for the training dataset and 20% of them for the testing dataset. This process is repeated for each host, and therefore we guarantee that 80% of each host (both normal and malware) is in the training set and 20% is in the testing set. Table 7.4 provides a summary of how many virtual hosts were on each partition of the dataset.

After learning the models by using the training dataset and testing in the testing datasets, the results are shown in Table 7.5. In addition to the same evaluation measures used for the other experiments, we also show two specific indicators to highlight known malware detection capability. They are reported in table 7.5. As $\mathbf{Kn_1}$ indicator shows, a significant proportion of infections were detected thanks to the prior information about them. $\mathbf{Kn_0}$ is non zero, suggesting the presence of virtual hosts correctly detected as infected thanks to the contribution of some information coming from other hosts infected by a different malware.

Table 7.4: Partitioning of 8,363 virtual hosts to detect known malware. Virtual hosts crafted from the same actual host are randomly divided 80% – 20% to form train and test datasets.

	TOTAL	Infected	Normal
Train	**6,694**	916	5,778
Test	**1,669**	214	1,455
TOTAL	**8,363**	**1,130**	**7,233**

7.7.4. DETECTING AN UNKNOWN THREAT

One interesting feature to evaluate in any malware detection algorithm is its capability to detect unknown malware. This means that the malware in the testing dataset must

Table 7.5: Known malware detection results. $\mathbf{Kn_1}$ is the proportion of true positives due to the same malware. $\mathbf{Kn_0}$ is the proportion of true positives due to different malware.

$\mathbf{Kn_1}$	$\mathbf{Kn_0}$	Acc	FPR	F1-score
0.962	0.038	0.997	0.002	0.988

not appear, in any way, in the training dataset. To check that property we create the train and test datasets differently. Since it is difficult to guarantee that the same malware does not appear in the testing set, we separate the complete dataset by malware family. This is possible because the CTU-13 dataset is split in scenarios. We selected the CTU-13 dataset scenarios 1, 2, 3, 4, 5, 9, 10, 11, 13 for training, and the CTU-13 dataset scenarios 6, 7, 8, 12, and datasets CTU 260-1, 261-1 for testing. By dividing the data in this way, we avoid using the same malware in the training and the testing datasets. Table 7.7 shows the virtual hosts partitioning in train and test dataset regarding their status of infected or normal, and table 7.8 summarises detection performance achieved by our algorithm.

Table 7.6 shows, for each malware in the test set, how much each malware in the training set contributed to its detection. That is, how a virtual host infected by malware in the testing set appeared as the closest neighbour of a malware virtual host in the training set. As already mentioned, train dataset contains information about nine malware - i.e. Neris in three configurations, Rbot in four configurations, Virut in two configurations - and test dataset contains five unknown malware - i.e. Menti, Sugou, Murlo, NSIS.ay, and probably Dridex and TrickBot. It is interesting to notice that only one malware is responsible for 100% of all correct decisions: Rbot in scenario 3 (see section 7.6). Its contribution consists in only two virtual hosts which appear in all the 214 virtual hosts correctly detected as infected. All those hosts are characterised by a peculiar behaviour: they are performing a scan on port 22 (SSH standard port). That makes sense since a port scanning is a usual preliminary malware activity.

Table 7.6: When correctly detecting malware in test dataset (first column), the decision has been made thanks to Rbot malware in train dataset. Parentheses include the scenario or dataset each malware is included, see Section 7.6 for further information.

probably Dridex (260-1)	**Rbot (3)**
probably TrickBot (261-1)	**Rbot (3)**
Menti (6)	**Rbot (3)**
Sogou (7)	**Rbot (3)**
Murlo (8)	**Rbot (3)**
NSIS.ay (12)	**Rbot (3)**

Table 7.7: Partitioning of 8363 virtual hosts for unknown malware detection. Virtual hosts from CTU-13 scenarios 6, 7, 8, 12, and CTU datasets 260-1, 261-1, form the test dataset. The remaining virtual hosts form the train dataset.

	TOTAL	Infected	Normal
Train	**6,266**	916	5,350
Test	**2,097**	214	1,883
TOTAL	**8,363**	**1,130**	**7,233**

Table 7.8: Unknown malware detection results.

TP	TN	FP	FN	Acc	FPR	F1-score
161	1,869	14	53	0.968	0.007	0.828

7.7.5. COMPARISON WITH BLACKLIST DETECTION

In this section, we compare our algorithm with a non-behavioural reputation system which employs a blacklist to build up its detection rules. Blacklists are a widely used tool for detection which consists of a list of entries known to be malicious. Those entries can be IP address in case of host detection, email address in case of spam detection, etc.. Network security operators keep blacklists updated by including knowledge coming from the observations of their network and integrating information coming from other networks sometimes offered by companies who provide blacklisting services. In this work, we focus on host detection. Therefore a candidate host is classified as infected when it attempts to communicate with any blacklisted entry.

This non-behavioural alternative leverages the same information of our proposed system, namely a dataset made of both normal and infected hosts. We filled the blacklist with all the destination ip addresses of network flows sent by an infected host in the dataset. Furthermore, we removed all the destination ip addresses of network flows sent by any trusted host from the blacklist. By doing so, the blacklist is only filled with the address of hosts which are communicating with known infected machines and with none of the known benign machines. Once the blacklist is filled, this system classifies a candidate host as infected if it tries to connect to any blacklisted entry. Note that blacklists are based on an attribute that our framework does not leverage at all: the destination ip address of the network flows.

As a first experiment, we used the cross-validation dataset described in section 7.7.2. We took each fold as the training dataset to fill the blacklist, and we used it to evaluate virtual hosts in the remaining folds. Table 7.9 the results for each round. Table 7.10 shows the result of our detection algorithm in the same experimental setup. It is important to highlight that here we used each fold as training dataset and the remaining as evaluation dataset at each round, which is a complementary setting to the experiment discussed in section 7.7.2 where one fold is used as evaluation dataset and the remaining as training. Results show a significant difference in performance both regarding false positives and, even more clearly, regarding false negatives.

Table 7.11 highlights a direct comparison among several alternative techniques including three baselines: always infected, always normal, and coin toss. Always Infected classifies each host as infected. Similarly, Always Normal classifies each host as normal. Coin toss classifies a host by tossing a fair coin and deciding for normal if heads and infected if tails. Results for every alternative are per-fold-averaged. We used the same settings summarised in table 7.7 to fill the blacklist and evaluate performance on unknown malware. Results are shown in table 7.12 with the already mentioned three baselines. Our behavioural based alternative outperforms the blacklist-based alternative in both experiments.

Table 7.9: Blacklist 5-fold cross validated detection performance.

	TP	TN	FP	FN	Acc	FPR	F1-score
F1	559	5,738	48	344	0.941	0.008	0.740
F2	558	5,731	55	345	0.940	0.010	0.736
F3	560	5,732	54	343	0.941	0.009	0.738
F4	562	5,734	52	341	0.941	0.009	0.741
F5	556	5,753	35	352	0.942	0.006	0.742
AVG	559	5,737	48	345	0.941	0.008	0.740

Table 7.10: 5-fold cross validation results in the blacklist setting. Discrete quantities (TP, TN, FP, FN) and measurements (Acc, FPR, F1-score) are averaged per-fold.

TP	TN	FP	FN	Acc	FPR	F1-score
892	5,750	36	11	0.993	0.006	0.974

Table 7.11: Cross validated detection performance comparison. PDA represents the proposed detection algorithm. Blacklist is the main alternative algorithm. Always Infected, Always Normal, and Coin Toss represent three different baselines.

	Acc	FPR	F1-score
PDA	0.993	0.006	0.974
BLACKLIST	0.941	0.008	0.740
ALWAYS INFECTED	0.179	1.000	0.238
ALWAYS NORMAL	0.865	0.000	0.000
COIN TOSS	0.508	0.492	0.218

Table 7.12: Detection performance comparison on detecting unknown malware. See Table 7.11 for further informations.

	Acc	FPR	F1-score
PDA	0.968	0.007	0.828
BLACKLIST	0.897	0.010	0.122
ALWAYS INFECTED	0.102	1.000	0.185
ALWAYS NORMAL	0.898	0.000	0.000
COIN TOSS	0.509	0.484	0.157

7.8. OBTAINING ACTIONABLE INSIGHTS

In scenario eight we have host 147.32.84.165 infected by a malware called Murlo (see Section 7.6). Such a host comprises a set of virtual hosts, each one explaining part of its behaviour (section 7.2 explains how it is possible to extract virtual hosts from actual hosts). We focus on one virtual host extracted from the mentioned host infected by Murlo, and we will denote the virtual hosts using the IP address of the actual host followed by a fifth decimal number. The fifth decimal number serves as the unique identifier for one of the extracted virtual hosts.

In our experiment virtual host 147.32.84.165.3 in scenario eight is classified as malicious because its closest virtual host is 147.32.84.165.1 in scenario 3, coming from the actual host 147.32.84.165 known as infected by RBot. Figure 7.7 shows the PDA for 147.32.84.165.1, with label LBTCP standing for Long and Big TCP flow, label SSTCP standing for Short and Small TCP flow, LSTCP standing for Long and Small TCP flow, and SBTCP standing for Short and Big TCP flow. A short flow means that its duration is less than 0.276 milliseconds otherwise it is intended as long. A small flow means that its size is less than 101.5 bytes.

The most occurring symptom in the profile for the PDA given a sample of flows coming from 147.32.84.165.3 is (1, SSTCP), occurring 64% of the times. This means that most of the times, 147.32.84.165.3 seems to fire the loop transition in state 1 or, in other words, it seems to spend most of the time in sending short and small TCP packets. Furthermore, when looking at the variation over time in profile distance between virtual hosts 147.32.84.165.3 and 147.32.84.165.1 using PDA of Figure 7.7, it seems that after 650 flows the profile distance stabilises below the 0.2 level. By limiting our study to the initial 650 flows sent by 147.32.84.165, it is possible to recognise the ones are characterising the mentioned symptom immediately:

```
0,TCP,147.32.84.165,1142,124.146.223.205,22,1,62
0,TCP,147.32.84.165,1143,124.146.223.206,22,1,62
0,TCP,147.32.84.165,1144,124.146.223.207,22,1,62
0,TCP,147.32.84.165,1145,124.146.223.208,22,1,62

0,TCP,147.32.84.165,2042,147.32.80.226,135,1,62
0,TCP,147.32.84.165,2043,192.168.1.226,135,1,62
0,TCP,147.32.84.165,2049,147.32.80.228,135,1,62
0,TCP,147.32.84.165,2048,192.168.1.228,135,1,62
```

Here we see an excerpt made of four short, and small TCP flows sent from the same host to different hosts at the same port 135. That is a particular port dedicated to RPC calls in client/server applications, also known as involved in several security threats [157]. We know host 147.32.84.165 in scenario eight as infected by Murlo malware, and tour framework was able to recognise its port scanning after having learned that type of behaviour from the Rbot malware.

In scenario eight we have a trusted virtual host, called 147.32.84.164.15, which has been classified as infected because of its closest neighbour is 147.32.84.206.12 in scenario nine, infected by the Neris malware. Apparently, the detection algorithm thinks that 147.32.84.164.15 is behaving as a host infected by Neris. Figure 7.8 shows the PDA of

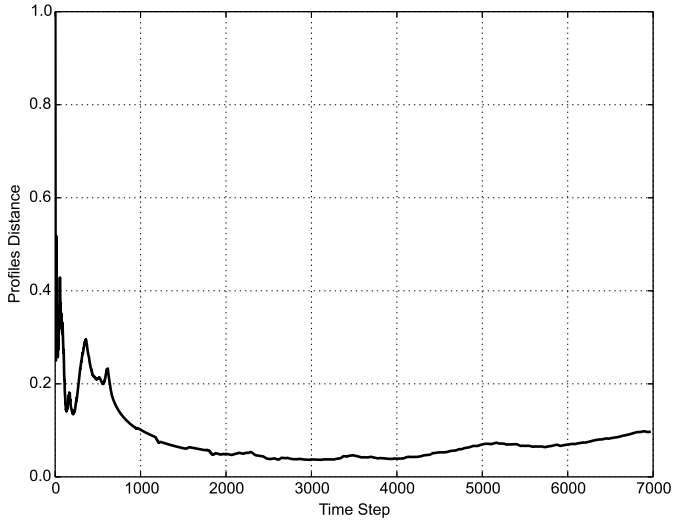


Figure 7.5: Profile distance over time between virtual hosts 147.32.84.165.3 in scenario 8 and 147.32.84.165.1 in scenario 3 using PDA in Figure 7.7. After 650 flows profile distance never hit the 0.2 threshold anymore.

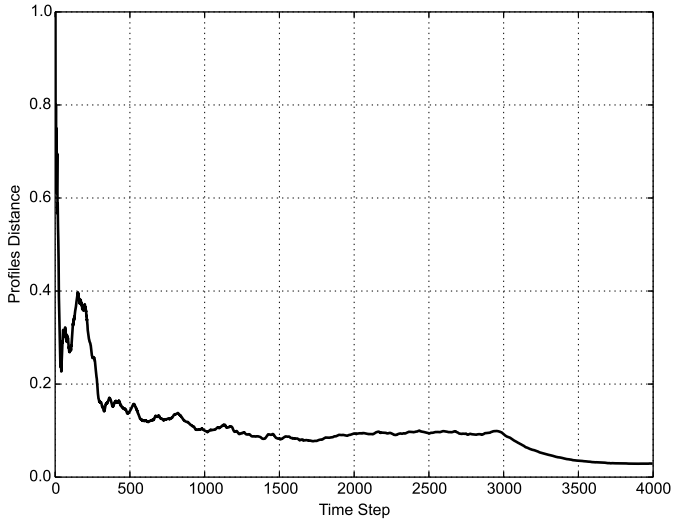


Figure 7.6: Profile distance over time between virtual hosts 147.32.84.164.15 in scenario 8 and 147.32.84.206.12 in scenario 9 using PDA in Figure 7.8. After 300 flows profile distance never hit the 0.2 threshold anymore.

the closest neighbor, where label SSUDP stands for short and Single datagram UDP flow, label BSUDP stands for Big and Single datagram UDP flow, BMUDP stands for Bing

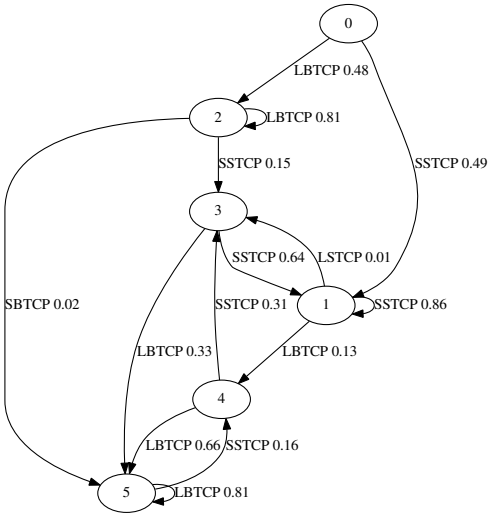


Figure 7.7: PDA representing virtual host 147.32.84.165.1 in scenario 3. Label LBTCP stands for Long and Big TCP flow, label SSTCP stands for short and small TCP flow, LSTCP stands for long and small TCP flow, and SBTCP stands for Short and Big TCP flow. Short flow means that its duration is less than 0.276 milliseconds otherwise it is meant as long. A small flow means that its size is greater than 101.5 bytes. Transitions to the sinking state have been omitted to improve readability.

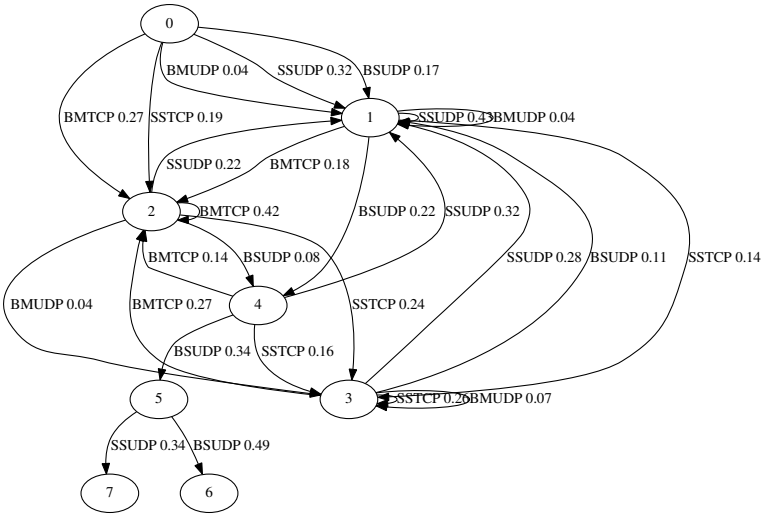


Figure 7.8: PDA representing virtual host 147.32.84.206.12 in scenario 9. Label SSUDP stands for short and Single datagram UDP flow, label BSUDP stands for Big and Single datagram UDP flow, BMUDP stands for Bing and Multi datagram UDP flow, SSTCP stands for Short and Single packet TCP flow, and BMTCP stands for Big and Multi packet TCP flow. Small flow means that its size is less than 75 bytes otherwise it is meant as big. Transitions to the sinking state have been omitted to improve readability.

and Multi datagram UDP flow, SSTCP stands for Short and Single packet TCP flow, and BMTCP stands for Big and Multi packet TCP flow. A small flow means that its size is less than 75 bytes otherwise it is labeled as big. When looking at the profile for that PDA given a sample of flows coming from 147.32.84.164.15, we realise that the most occurring symptoms are (1, SSUDP) and (2, BMTCP), which occur 59% of the times. It means that quite often, 147.32.84.164.15 sends short UDP flows or opens multi-packet TCP connections.

We can reduce the number of flows to investigate by looking at the variation of profiles distance over time and look up to that moment after which it stably lies below a reasonable threshold value. Figure 7.6 shows that it is enough to explore the first 300 flows of both virtual hosts. By looking at the data, we observe similar instances of those characteristic behaviours in both virtual hosts. Here are few instances of flows coming from host 147.32.84.164.15 that correspond to the symptom (1, SSUDP):

```
0.000,UDP,147.32.84.164,42395,147.32.80.9,53,1,74
0.000,UDP,147.32.84.164,53140,147.32.80.9,53,1,74
0.000,UDP,147.32.84.164,50171,147.32.80.9,53,1,74
```

And here we have few instances of flows coming from the same host, that correspond to the symptom (2, BMTCP):

```
3,TCP,147.32.84.164,60124,146.102.14.10,443,9,1665
0,TCP,147.32.84.164,60125,46.102.14.10,443,5,612
3,TCP,147.32.84.164,60124,146.102.14.10,443,10,6501
```

By moving from flow level down to the underlying packets capture, we see that those single UDP flows correspond to normal datagrams involved in some steps of a DNS resolution query as port 53 may suggest - in this case, it is a query for `www.ceknito.cz`, a Czech video sharing website. The TCP flows characterizing symptom (2, BMTCP) seems due to normal and legitimate web surfing activity - in this case, a request for `www.insis.vse.cz`, the official website of the Integrated Study Information System of the University of Economics in Prague. What seems to emerge from this analysis is that the proposed detection algorithm matched a trusted host's legitimate behaviour with an infected host's legitimate behaviour.

7.9. LIMITATIONS

The above false positive example gives us the opportunity to discuss the limitations of the proposed method. A first limitation comes from the decision of working with network flows since the payload is not inspected and therefore it is harder to distinguish between legitimate and malicious requests. As such, a framework such as the one we propose should be used in conjunction with existing rule-based intrusion detection systems that do inspect such data. As we show in this work, behavioral malware detection is a valuable addition to such systems.

Besides the data type, the above example highlights a potential weak point in the data abstraction part of the proposed method (see Section 7.2). The framework discussed

Table 7.13: Detection performance when posponing the dara abstraction of the malicious virtual hosts at test time. The experiment setting is the same of Section 7.7.2, see Table 7.3.

TP	TN	FP	FN	Acc	FPR	F1-score
217	1372	73	8	0.951	0.051	0.841

in this chapter, composed of PDAs, PDA learning algorithm, symptoms, and profiles, is based on the symbolic representation obtained from the abstraction process. A botnet developer could, in principle, design a C&C communication protocol such that it is capable of circumventing the data abstractor by, for instance, letting an infected host produce mainly the same type of network flows and, therefore, the same symbols. Although we are no botnet authors, we are confident that making such a change is far from easy, as we have yet to encounter such a bot. Moreover, a connection with identical flows is suspicious, and numerous existing methods can detect such behaviour, e.g., using entropy.

Since our abstraction is fixed, however, if a botnet developer somehow gains knowledge of the bounds we use, (s)he could in theory modify the flows slightly to change the generated symbols and avoid detection. A promising direction for future work is therefore to increase the flexibility of this symbolic abstration. We have performed a preliminary analysis of such an approach by abstracting the data at test time when creating a profile for a malicious PDA. In this way, deviations in the flow data values do not influence the detection capabilities. There is however a cost in terms of increased false positives. Table 7.13 shows the results of this approach when repeating the experiment from Table 7.3. There is a significant increase in the number of false positives, i.e., hosts that are now matched with malicious PDAs due to the flexible flow abstraction. But the number of true positives remains nearly the same. In future work, we aim to discover ways to decrease the number of false positives, for instance using additional filters, combining it with a fixed abstraction, or creating a different way to introduce flexibility.

Notice that this type of flow modification only fools the malicious PDA profiles from the same bot. It does not influence the distance to the profiles of local PDAs, or any of the other botnet PDAs. Thus, in order to evade detection, a bot should modify its behavior to match one of the local hosts, and to be different from existing bots. Since hosts behave differently in different networks, and there are many botnet profiles, this will be very hard if not impossible to achieve.

8

Conclusions

This thesis describes years of research into the automatic profiling of a host's behaviour observed in network captures for detecting malicious activity. The thesis details a methodology that starts from Netflow captures and ends with automata: the model of choice for describing software dynamics. This methodology makes classifying network participants in terms of malicious behaviour and legitimate behaviour classes possible. Automata are automatically inferred from Netflow captures and used as network agnostic generators for network-specific behavioral profiles of the hosts. This final chapter of the thesis consists of two main sections: Section 8.1 addresses the main contributions of this research, and Section 8.2 discusses future improvements.

8.1. CONTRIBUTIONS

The principal contribution of this thesis is an automata-based methodology for detecting network intrusions. The proposed methodology is host-centric and divided into two distinct phases: training and testing.

One main goal of the training phase is to build and maintain a library of automata. Automata are either automatically inferred from the network traffic captured in the Network Under Observation (NUO) or imported from other networks. Cross-network abstraction is possible because streams of Netflows are translated into generic sequences of symbols. By preserving the same components as those responsible for such a symbolic translation, automata are freely sharable across networks, as is the case with the signatures for Intrusion Detection/Prevention Systems or YARA rules[1] in malware research.

The other main goal of the training phase is to build and maintain a library of behaviour profiles. Automata collected in the library consume sequences of symbols as inputs and, as outputs, produce what we call symptoms. A symptom is a couple composed of an input symbol and an inner state of an automaton. Symptoms collected from a host are grouped in a multiset called the behaviour profile of a host. Behaviour profiles are specific to the NUO: they are not meant to be shared across networks because they express the behaviour of the hosts in the NUO. Behaviour profiles might represent either malicious or legitimate activities, according to the security state of the host that produced the Netflows. The training phase relies on labelled data: it is necessary to know whether network captures were taken from a compromised or legitimate host.

The testing phase aims to spot malicious activities in Netflow captures when the security status of the hosts producing the captures is unknown. During testing, Netflow streams are translated into multiple sequences of symbols by applying the same procedure and components as those used for the training phase. Later, those sequences are consumed by the automata in the library, and new behavior profiles are produced for the hosts. A new alarm is raised whenever any of those profiles matches a malicious behaviour profile included in the profiles library.

[1] YARA is both a tool for classifying software artefacts and a language for expressing rules for matching patterns. For details, see [158].

Because the proposed methodology addresses encryption unawareness, scalability, ease of maintenance of the knowledge base, and detection of the unknown, it enables the development of advanced NIDSs.

Adopting Netflows as the network data's source of choice comes with two interesting properties: encryption unawareness and scalability. Because Netflows group network packets by various fields unrelated to packet contents, such as the number of packets within a flow, payload encryption does not affect the performance of an NIDS implementing the proposed methodology. Scalability is improved because Netflows are an aggregated type of network data (each Netflow can summarise many packets); therefore, scanning each packet in a network capture is unnecessary. Scalability is further improved by adopting a host-centric perspective. That is, the focus is on expressing overall judgements about the security state of a host, not on generating a verdict on each packet, as happens with most NIDS.

Introducing automata as network-agnostic generators of network-specific behaviour profiles for hosts comes with a few advantages. First, compared to common rule-based or signature-based NIDSs, each single automaton can express possibly many detection rules. Second, in contrast to what happens with rules and signatures designed by domain experts, automata are automatically learnable from Netflow captures. In this thesis, we show how this is done by leveraging different learning algorithms for different types of target automata. Third, learning automata with no human intervention alleviates another common weakness of NIDS: the management of knowledge bases. Because the knowledge base in the proposed methodology is based on a library of automata, its maintenance and update do not require expert knowledge or significant effort. Finally, automata-learning algorithms advocated for the proposed methodology exhibit a sufficient level of abstraction power to enable the learned automata to catch the behaviour not directly expressed in the training data. This property is attractive because it makes the proposed methodology capable of spotting threats not mapped in the knowledge base. That overcomes a known limit of the knowledge-based NIDSs, which are prone to false negatives.

Further contributions can be divided into two different classes: automata-learning-related and automata-based-NIDSs.

A first class addresses several relevant problems attributed to learning automata from network data. In Chapter 3, we explore the possibility of postponing the symbolic abstraction of the Netflows during learning for exploiting local dependencies in the data. Such symbolic abstraction, also called labelling, is one of the stages of the proposed methodology implemented in the Labeler component. Labelling involves translating raw data into abstract symbols, a mandatory step in learning automata. Usually, the labelling process is designed as a pre-learning step, where the training data is mapped into symbols according to some global criterion. However, proceeding in this manner carries the risk of missing short-term dependencies. To model these data relationships, a new type of automaton, called Regression Automaton (RA), is introduced in Chapter 3.

In RAs, the transitions from one state to another are controlled by guards based on non-overlapping intervals. Furthermore, an algorithm for learning RAs from data, Regression Automata Identifier (RAI), is presented and discussed. RAI is innovative because, to the author's knowledge, it is the first algorithm capable of learning transition guards by following a bottom-up approach. We tested RAI against six alternative techniques in four different prediction scenarios. When there are local dependencies, such as predicting a distorted sinus wave, RAI models proved to have the best performance. In scenarios that expose a periodic signal with fewer local dependencies, such as predicting an unusual sinus wave, RAI models proved that performance aligns with the alternative techniques.

In Chapter 4, we address the problem of obtaining sequences of consecutive symbols from a potentially infinite stream. This step is implemented in the Sequencer component of the proposed methodology. Stream sequencing is required to learn automata from network data because automata are computational models capable of processing words, and network data comes as streams of packets or flows. One of the most common techniques for solving this problem is sliding a fixed size window along the stream. When it is impossible to devise a criterion from the application domain, the sliding window technique is the only choice. However, as the experiments show, the sliding window leads to more complex automata in most cases. We propose a new technique aimed at obtaining words by segmenting a stream. Our technique leverages a knowledge base of known words belonging to the target automaton's language to build a language model able to find other segmentation points. The assumption of having a few known words is considered acceptable in network data. Tests on 48 synthetic target models of different sizes and typologies showed that the proposed technique outperforms sliding windows in 33 cases by leveraging a base of only 200 words, corresponding to 5% of correct words in the training data. With a base of 600 words, corresponding to 15% of the correct words from the training data, the proposed technique outperforms the sliding window on 36 problems out of 48. Experiments also show that the proposed technique tends to be most performant when the target model is deterministic or has high transition sparsity. As shown in Chapter 7 and Chapter 6, suitable models for profiling of hosts' behaviours are deterministic and transition-sparse.

In Chapter 5, we discuss the problem of collecting the right amount of data needed to learn a probabilistic automaton. Ideally, the goal is to collect the most minor data required to identify all states in the target model. The main contribution of this chapter is to introduce a method for recognising, in real time, the so-called points of change. A point of change happens when including a word in the training set introduces an unforeseen behaviour. An automaton learned before a point of change tends to expose structural differences compared to the one learned after the point of change. The proposed method relies on a newly introduced freshness measure to catch those structural differences within an augmented prefix tree: a canonical automaton usually learned in the first steps of most automata-learning algorithms. The analyst can identify the points of change after analysing a freshness plot. In the same chapter, we introduce Hoeffding-bound signals to indicate whether the collected data is sufficient to estimating the transition probabilities of a probabilistic automaton. Experimental results on real Netflow

captures in different network scenarios show that the combination of freshness and Hoeffding signals allows the analyst to collect, in most cases, significantly smaller training samples without affecting the quality of the learned models. Automating the points of change detection in freshness and Hoeffding plots allows the analyst to automatically detect the time at which the Netflows collected from a host constitute enough data for creating a meaningful behaviour model.

The second class of contributions provides details of different methods of using automata to detect network intrusions and to analyse network data. We discuss the first implementation of the proposed methodology in Chapter 7. Chapter 7 introduces a new implementation of the proposed methodology, which uses Probabilistic Deterministic Automata (PDA). One of the main contributions presented in the chapter consists of defining a host behaviour profile as a probability distribution over the so-called symptoms: the network-specific behavioural features generated by a PDA. The chapter introduces a new Profiles Matcher. This component is based on the definition of a chi-square distance between hosts' profiles and a nearest-neighbour approach for classifying a host's behaviour. The proposed implementation has been extensively tested on different publicly available datasets of Netflow captures. It showed an F1-score greater than 95%, with a shallow rate of false positives (0.004) in a cross-validated setting. The proposed implementation can also deal with unknown threats—that is, infections it has no prior knowledge about—because it performed adequately in an experiment designed to measure detection; that is, it produced an F1 score of 0.828 with the false-positives rate of 0.007. Finally, the proposed implementation has been compared with a blacklist-based detector. Our system outperformed the blacklist detector in accuracy, false positives rate, and F1 score. The chapter also describes how to simplify the analysis of a Netflow capture by leveraging the proposed implementation. Indeed, thanks to the interpretation of the behaviour profiles, an analyst can reduce the number of Netflows to be analysed by quickly converging the instances responsible for a detection.

Chapter 6 introduces a second implementation of the proposed methodology: BASTA, which uses an RTI+ Learner component to identify Probabilistic Deterministic Real-Time Automata (PDRTAs). PDRTAs are the models chosen to describe hosts' behaviours. Furthermore, we introduce a simple but effective Labeler component to obtain timed strings from Netflows using attribute mappings and the sliding-window Sequencer. Another contribution discussed in the chapter is the presentation of two alternative implementations for the Profile Matcher component: the error-based matcher and the fingerprint-based matcher. The error-based matcher provides better results on noiseless data, and simple fingerprints work better in noisy settings. This is likely due to incorrectly estimated expected counts, and it suggests that an error-based matcher is better in noise-free settings, where the network manager can isolate the traffic by using, for instance, IP destinations. When this results in too little data, using the fingerprinting matcher on the noisy intertwined traffic is better. Both matchers can effectively detect known infections, as shown in our experiments on single scenarios from a public dataset of real Netflow captures. In fact, experiments aimed at detecting known infections showed that BASTA detected nearly all infected hosts and produced very few false

positives. By mixing the traffic from and to different hosts, BASTA retained its good performance in detecting a known threat. Finally, in experiments to understand the capabilities of detecting unknown threats, BASTA proved effective even with a raised number of false positives. The author argues that those results demonstrate the effectiveness of the proposed methodology. BASTA, or other potential implementations of the proposed methodology, can be integrated into a state-of-the-art network analyser to complement different types of IoC-based detection engines.

8.2. FUTURE WORK

There are many directions for future work, and this section aims to highlight some of them. The discussion is organised into two parts, the first focusing on viable future research activities that attempt to solve relevant problems arising from learning automata from network data.

RAI, the Learner for Regression Automata discussed in Chapter 3, relies on a neighbourhood-based statistical test to decide whether to merge two states. Although the statistical test proved successful overall, it is not perfect: that is, it might cause wrong decisions, affecting the final quality of the learned model. Investigating the root causes of those mistakes might foster the design or adoption of more performant statistical tests. A further line of research into RAI and RAs consists of extending both the model and the learning algorithms to operate on multivariate data, such as discrete and continuous fields. By removing the Labeler, such an extension could lead to a change in the methodology because Netflows are composed of, essentially, discrete and continuous fields. The labelling would get integrated into the Learner, and the added benefit would be the addition of a capability for modelling local relationships within the data.

The semi-supervised Segmenter, discussed in Chapter 4, has yet to be evaluated on Netflow captures due to the lack of a known target model for evaluating the results. The author considers it crucial to integrate the Segmenter into an implementation of the proposed methodology, such as the BASTA system. In addition, it would be interesting to compare the models learned fromNetflowdata against a time-threshold–based alternative because a common strategy for segmenting a stream of network data is to use time thresholds. A possible evaluation would be found on the learned model's contributions to the detection performances.

As discussed in Chapter 5, assessing whether enough data has been collected to learn automaton is mainly done by an analyst. The analyst looks at the freshness and Hoeffding plots and recognises when a point of change corresponds to a remarkable transformation of the hypothesis. A possible line of research on this topic would be to automate such a process by developing an algorithmic solution capable of detecting points of change and, therefore, understanding when to stop data collection. A viable option would be to use a Kullback-Leibler divergence to measure a broader range of change points, even the slow drifts in the occurrence of existing behaviours that are harder to identify for a human.

The second direction for future work is about viable options for implementing tools for detecting network intrusions and analysing network data using automata. For the system described in Chapter 7, implementing our methodology, future research activities could be aimed at evaluating the possibilities the system offers and the limitations it might be subject to. Testing the use case when PDAs learned in other networks are imported into the Automata Library would be interesting. The author considers that use case crucial to accepting PDAs as a behavioural indicator for network threats. Another exciting line of research is to study how difficult it is to evade detection when the threat actors know the model used for generating the behavioral profiles. That line of research could improve the methodology resilience by making the Data Abstraction phase more robust to opportunistic changes in attribute values.

The author envisions at least two possible options as a future work on the BASTA system discussed in Chapter 6. First, it would be interesting to evaluate the performance of BASTA on a vast real network, such as the TU Delft campus network. Second, BASTA did not show the same level of performance on noisy input data: that is, mixed malicious and legitimate flow sequences coming from the same host. A possible solution, and a subject of future work, would be to design a new Profiles Generator component to deal specifically with noisy data. That Profiles Generator could apply dynamic programming techniques to filter out noise.

BIBLIOGRAPHY

[1] CrowdStrike, *2021 Global Threat Report*, Tech. Rep. (CrowdStrike Holdings Inc., 2021).

[2] CrowdStrike, *2022 Global Threat Report*, Tech. Rep. (CrowdStrike Holdings Inc., 2022).

[3] Infoblox, *The Q4 2020 Cyberthreat Intelligence Report*, Tech. Rep. (Infoblox, 2021).

[4] United States Department of Homeland Security (DHS), Cybersecurity and Infrastructure Security Agency (CISA), and the United Kingdom's National Cyber Security Centre (NCSC), *Covid-19 exploited by malicious cyber actors,* https://us-cert.cisa.gov/ncas/alerts/aa20-099a (2020), last accessed: 2023-10-10.

[5] C. Castrillon, *This is the future of remote work in 2021,* https://www.forbes.com/sites/carolinecastrillon/2021/12/27/this-is-the-future-of-remote-work-in-2021/?sh=e1696971e1de (2020), last accessed: 2023-10-10.

[6] P. Fogarty, S. Frantz, J. Hirschfeld, S. Keating, E. Lafont, B. Lufkin, R. Mishael, V. Ponnavolu, M. Savage, and M. Turits, *Coronavirus: How the world of work may change forever,* https://www.bbc.com/worklife/article/20201023-coronavirus-how-will-the-pandemic-change-the-way-we-work (2020), last accessed: 2023-10-10.

[7] D. S. Unit, *Special Report: Ukraine*, Tech. Rep. (Microsoft, 2011).

[8] K. Grauer, W. Kueshner, and H. Updegrave, *The 2022 Crypto Crime Report*, Tech. Rep. (Chainalysis, 2022).

[9] A. Sally, *The State of Ransomware 2022*, Tech. Rep. (Sophos, 2022).

[10] M. Fuentes, F. Hacquebord, S. Hilt, I. Kenefick, V. Kropotov, R. McArdle, F. Mercês, and D. Sancho, *Modern Ransomware's Double Extortion Tactics and How to Protect Enterprises Against Them*, Tech. Rep. (Trendmicro, 2021).

[11] I. Lella, M. Theocharidou, E. Tsekmezoglou, A. Malatras, S. Garcia, and V. Valeros, *ENISA Threat Landscape for Supply Chain Attacks*, Tech. Rep. (ENISA, 2021).

[12] T. Brooks, J. Heathley, S. Kim, S. Parks, M. Reardon, H. Rohrbacher, B. Sahin, *et al.*, *Increasing Threats of Deepfake Identities*, Tech. Rep. (U.S. Department of Homeland Security, 2021).

[13] C. Douligeris, O. Raghimi, M. Barros Lourenço, L. Marinos, A. Sfakianakis, C. Doerr, J. Armin, M. Riccardi, W. Mees, N. Thaker, P. Stirparo, P. Samwel, P. Paganini, S. Adachi, S. Lingris, and T. Hemker, *ENISA Threat Landscape - 2020*, Tech. Rep. (ENISA, 2020).

[14] FireEye, *Deep Dive Into Cyber Reality*, Tech. Rep. (FireEye, 2021).

[15] FireEye, *M-TRENDS 2021*, Tech. Rep. (FireEye, 2021).

[16] C. Singleton, *X-Force Threat Intelligence Index 2021*, Tech. Rep. (IBM Security, 2021).

[17] K. Grauer and H. Updegrave, *The 2021 Crypto Crime Report*, Tech. Rep. (Chainalysis, 2021).

[18] Cybersecurity and Infrastructure Security Agency (CISA), *Weak security controls and practices routinely exploited for initial access,* https://www.cisa.gov/uscert/ncas/alerts/aa22-137a (2022), last accessed: 2023-10-10.

[19] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, and F. Ahmad, *Network intrusion detection system: A systematic study of machine learning and deep learning approaches,* Transactions on Emerging Telecommunications Technologies **32** (2021).

[20] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, *Intrusion detection system: A comprehensive review,* Journal of Network and Computer Applications **36** (2013).

[21] H. Liu and B. Lang, *Machine learning and deep learning methods for intrusion detection systems: A survey,* Applied Sciences **9** (2019).

[22] N. Habra, B. L. Charlier, A. Mounji, and I. Mathieu, *ASAX: Software architecture and rule-based language for universal audit trail analysis,* in *European Symposium on Research in Computer Security* (Springer, 1992).

[23] K. A. Jackson, D. H. DuBois, and C. A. Stallings, *An expert system application for network intrusion detection*, Tech. Rep. (National Institute of Standards and Technology / National Computer Security Center, 1991).

[24] V. Paxson, *Bro: a system for detecting network intruders in real-time,* in *Proceedings of the 7th USENIX Security Symposium* (USENIX Association, 1998).

[25] Y. F. Jou, F. Gong, C. Sargor, S. F. Wu, and W. R. Cleaveland, *Architecture design of a scalable intrusion detection system for the emerging network infrastructure,* Tech. Rep. (Department of Computer Science, North Carolina State University, 1997).

[26] S. E. Smaha, *Haystack: An intrusion detection system,* in *Fourth Aerospace Computer Security Applications Conference*, Vol. 44 (Orlando, FL, USA, 1988).

[27] L. T. Heberlein, G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber, *A network security monitor,* in *Proceedings of the 1990 IEEE Symposium On Research in Security and Privacy* (1990).

[28] M. M. Sebring, E. Shellhouse, M. E. Hanna, and R. A. Whitehurst, *Expert systems in intrusion detection: A case study,* in *Proceedings of the 11th National Computer Security Conference*, Vol. 1 (National Institute of Standards and Technology / National Computer Security Center, 1988).

[29] C. Dowel and P. Ramstedt, *The computer watch data reduction tool,* in *Proceedings of the 13th National Computer Security Conference*, Vol. 13 (1990).

[30] Y. Shen, K. Zheng, C. Wu, M. Zhang, X. Niu, and Y. Yang, *An ensemble method based on selection using bat algorithm for intrusion detection,* The Computer Journal (2018).

[31] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, *A deep learning approach to network intrusion detection,* IEEE transactions on emerging topics in computational intelligence **2** (2018).

[32] M. H. Ali, B. A. D. Al Mohammed, A. Ismail, and M. F. Zolkipli, *A new intrusion detection system based on fast learning network and particle swarm optimization,* IEEE Access **6** (2018).

[33] B. Yan and G. Han, *Effective feature extraction via stacked sparse autoencoder to improve intrusion detection system,* IEEE Access **6** (2018).

[34] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han, *Enhanced network anomaly detection based on deep neural networks,* IEEE access **6** (2018).

[35] M. Al-Qatf, Y. Lasheng, M. Al-Habib, and K. Al-Sabahi, *Deep learning approach combining sparse autoencoder with svm for network intrusion detection,* IEEE Access **6** (2018).

[36] N. Marir, H. Wang, G. Feng, B. Li, and M. Jia, *Distributed abnormal behavior detection approach based on deep belief network and ensemble svm using spark,* IEEE Access **6** (2018).

[37] H. Yao, D. Fu, P. Zhang, M. Li, and Y. Liu, *Msml: A novel multilevel semi-supervised machine learning framework for intrusion detection system,* IEEE Internet of Things Journal **6** (2019).

[38] X. Gao, C. Shan, C. Hu, Z. Niu, and Z. Liu, *An adaptive ensemble machine learning model for intrusion detection,* IEEE Access **7** (2019).

[39] G. Karatas, O. Demir, and O. K. Sahingoz, *Increasing the performance of machine learning-based idss on an imbalanced and up-to-date dataset,* IEEE Access **8** (2020).

[40] C. Yin, Y. Zhu, J. Fei, and X. He, *A deep learning approach for intrusion detection using recurrent neural networks,* IEEE Access **7** (2017).

[41] Y. Jia, M. Wang, and Y. Wang, *Network intrusion detection algorithm based on deep neural network,* IET Information Security **13** (2019).

[42] Z. Wang, *Deep learning-based intrusion detection with adversaries,* IEEE Access **6** (2018).

[43] C. Xu, J. Shen, X. Du, and F. Zhang, *An intrusion detection system using a deep neural network with gated recurrent units,* IEEE Access **6** (2018).

[44] D. Papamartzivanos, F. G. Mármol, and G. Kambourakis, *Introducing deep learning self-adaptive misuse network intrusion detection systems,* IEEE Access **7** (2019).

[45] F. A. Khan, A. Gumaei, A. Derhab, and A. Hussain, *A novel two-stage deep learning model for efficient network intrusion detection,* IEEE Access **7** (2019).

[46] Y. Xiao, C. Xing, T. Zhang, and Z. Zhao, *An intrusion detection model based on feature reduction and convolutional neural networks,* IEEE Access **7** (2019).

[47] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman, *Deep learning approach for intelligent intrusion detection system,* IEEE Access **7** (2019).

[48] P. Wei, Y. Li, Z. Zhang, T. Hu, Z. Li, and D. Liu, *An optimization method for intrusion detection classification model based on deep belief network,* IEEE Access **7** (2019).

[49] X. Zhang, J. Chen, Y. Zhou, L. Han, and J. Lin, *A multiple-layer representation learning model for network-based attack detection,* IEEE Access **7** (2019).

[50] R. K. Malaiya, D. Kwon, J. Kim, S. C. Suh, H. Kim, and I. Kim, *An empirical evaluation of deep learning for network anomaly detection,* in *2018 International Conference on Computing, Networking and Communications (ICNC)* (IEEE, 2018).

[51] K. Jiang, W. Wang, A. Wang, and H. Wu, *Network intrusion detection combined hybrid sampling with deep hierarchical network,* IEEE Access **8** (2020).

[52] Y. Yang, K. Zheng, B. Wu, Y. Yang, and X. Wang, *Network intrusion detection based on supervised adversarial variational auto-encoder with regularization,* IEEE Access **8** (2020).

[53] Y. Yu and N. Bian, *An intrusion detection method using few-shot learning,* IEEE Access **8** (2020).

[54] G. Andresini, A. Appice, N. D. Mauro, C. Loglisci, and D. Malerba, *Multi-channel deep feature learning for intrusion detection,* IEEE Access **8** (2020).

[55] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide* (Addison-Wesley Professional, 2005).

[56] D. Harel, *Statecharts: A visual formalism for complex systems,* Science of Computer Programming **8** (1987).

[57] S. Kumar and E. H. Spafford, *A pattern matching model for misuse intrusion detection,* in *Proceedings of the 17th National Computer Security Conference*, Vol. 1 (National Institute of Standards and Technology / National Computer Security Center, 1994).

[58] K. Jensen, *Coloured Petri Nets*, Vol. 1 (Springer Verlag, 1992).

[59] J. Esparza, M. Leucker, and M. Schlund, *Learning workflow petri nets,* Fundamenta Informaticae **113** (2011).

[60] K. Ilgun, *USTAT: A real-time intrusion detection system for unix,* in *Proceedings 1993 IEEE Computer Society Symposium on Research in Security and Privacy* (IEEE, 1993).

[61] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars* (Cambridge University Press, 2010).

[62] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, *Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains*, Tech. Rep. (Lockheed Martin Corporation, 2011).

[63] N. Brownlee, C. Mills, and G. Ruth, *RFC 2722 - traffic flow measurement: Architecture,* http://www.ietf.org/rfc/rfc2722.txt (1999), last accessed: 2023-10-11.

[64] J. Rajahalme, A. Conta, B. Carpenter, and S. Deering, *RFC 3697 - ipv6 flow label specification,* https://www.ietf.org/rfc/rfc3697.txt (2004), last accessed: 2023-10-11.

[65] J. Quittek, J. Zseby, B. Claise, and S. Zander, *RFC 3917 - ipfix requirements,* https://www.ietf.org/rfc/rfc3917.txt (2004), last accessed: 2023-10-11.

[66] J. Ham and S. Davidoff, *Network Forensics: Tracking Hackers through Cyberspace*, Vol. 2014 (Prentice Hall, 2012).

[67] N. Shah, *Keeping up with the performance demands of encrypted web traffic,* https://www.fortinet.com/blog/industry-trends/keeping-up-with-performance-demands-of-encrypted-web-traffic (2020), last accessed: 2023-10-10.

[68] R. Messier, *Network Forensics* (Wiley Publishing, 2017).

[69] T. Steffens, *Attribution of Advanced Persistent Threats - How to Identify the Actors Behind Cyber-Espionage* (Springer, 2020).

[70] D. J. Bianco, *The pyramid of pain,* https://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html (2013), last accessed: 2023-10-10.

[71] O. Vlcek, *Behavior shield: our newest behavioral analysis technology,* https://blog.avast.com/behavior-shield-our-newest-behavioral-analysis-technology (2017), last accessed: 2023-10-10.

[72] McAfee Corporation, *What is advanced endpoint,* https://www.mcafee.com/enterprise/en-us/security-awareness/endpoint/what-is-advanced-endpoint-protection.html (2019), last accessed: 2023-10-11.

[73] Kaspersky Lab, *Behavior-based protection,* https://www.kaspersky.com/enterprise-security/wiki-section/products/behavior-based-protection (2021), last accessed: 2023-10-11.

[74] Microsoft Corporation, *Client behavioral blocking,* https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/client-behavioral-blocking (2021), last accessed: 2023-10-11.

[75] Yoroi, *Yomi hunter joined the virustotal sandbox program!* https://yoroi.company/announcement/yomi-hunter-joined-the-virustotal-sandbox-program (2019), last accessed: 2023-10-11.

[76] Information Sciences Institute University of Southern California, *RFC 793 - transmission control protocol,* https://www.ietf.org/rfc/rfc793.txt (1981), last accessed: 2023-10-11.

[77] Z. C. Lipton, *The mythos of model interpretability,* ACM Queue **16** (2018).

[78] M. Bateson and P. Martin, *Measuring Behaviour: an Introductory Guide* (Cambridge University Press, 2021).

[79] D. R. MacNulty, L. D. Mech, and D. W. Smith, *A proposed ethogram of large-carnivore predatory behavior, exemplified by the wolf,* Journal of Mammalogy **88** (2007).

[80] A. D. De Groot, *Methodology: Foundations of inference and research in the behavioral sciences*, Vol. 6 (Walter de Gruyter GmbH & Co KG, 2020).

[81] S. E. Verwer, *Efficient identification of timed automata: theory and practice*, Ph.D. thesis, Delft University of Technology (2010).

[82] E. M. Gold, *Language identification in the limit,* Information and Control **10** (1967).

[83] D. Angluin, *Identifying Languages from Stochastic Examples,* Technical report (Yale University, Department of Computer Science, 1988).

[84] L. G. Valiant, *A theory of the learnable,* Communications of the ACM **27** (1984).

[85] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory* (MIT Press, 1994).

[86] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, *Learnability and the vapnik-chervonenkis dimension,* Journal of the ACM (JACM) **36** (1989).

[87] Crossmuller, *The benefits of finite state machines in industrial automation,* https://www.crossmuller.com.au/news/the-benefits-of-finite-state-machines-in-industrial-automation/ (2020), last accessed: 2023-10-26.

[88] J. B. Halpern, *The use of finite state machine representation in elevator control sequence specifications,* in *Elevator Technology: Proceedings of ELEVCON '92* (International Association of Elevator Engineers, 1992).

[89] S. M. Ross, *Introduction to Probability Models* (Academic Press, 1997).

[90] L. R. Rabiner, *A tutorial on hidden markov models and selected applications in speech recognition,* Proceedings of the IEEE **77** (1989).

[91] P. Dupont, F. Denis, and Y. Esposito, *Links between probabilistic automata and hidden markov models: Probability distributions, learning models and induction algorithms,* Pattern Recognition **38** (2005).

[92] R. Alur and D. L. Dill, *A theory of timed automata,* Theoretical Computer Science **126** (1994).

[93] C. Dima, *Real-time automata,* Journal of Automata, Languages and Combinatorics **6** (2001).

[94] K. J. Lang, B. A. Pearlmutter, and R. A. Price, *Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm,* in *Grammatical Inference* (Springer Berlin Heidelberg, 1998).

[95] R. C. Carrasco and J. Oncina, *Learning stochastic regular grammars by means of a state merging method,* in *Grammatical Inference and Applications* (Springer Berlin Heidelberg, 1994).

[96] G. Pellegrino, Q. Lin, C. Hammerschmidt, and S. Verwer, *Learning deterministic finite automata from infinite alphabets,* in *International Conference on Grammatical Inference*, Vol. 57 (Proceedings of Machine Learning Research, 2017).

[97] G. Weiss, Y. Goldberg, and E. Yahav, *Extracting automata from recurrent neural networks using queries and counterexamples,* arXiv preprint arXiv:1711.09576 (2017).

[98] C. A. Hammerschmidt, Q. Lin, S. Verwer, and R. State, *Interpreting finite automata for sequential data,* arXiv preprint arXiv:1611.07100 (2016).

[99] S. Verwer, M. de Weerdt, and C. Witteveen, *Efficiently identifying deterministic real-time automata from labeled data,* Machine Learning **86** (2012).

[100] Q. Lin, C. Hammerschmidt, G. Pellegrino, and S. Verwer, *Short-term time series forecasting with regression automata,* in *SIGKDD 2016 Workshop on Mining and Learning from Time Series (MiLeTS)* (2016).

[101] J. Schmidt, A. Ghorbani, A. Hapfelmeier, and S. Kramer, *Learning probabilistic real-time automata from multi-attribute event logs,* Intelligent Data Analysis - Dynamic Networks and Knowledge Discovery **17** (2013).

[102] O. Niggemann, B. Stein, A. Maier, A. Vodenčarevič, and H. K. Büning, *Learning behavior models for hybrid timed systems,* in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12 (2012).

[103] N. Walkinshaw, R. Taylor, and J. Derrick, *Inferring extended finite state machine models from software executions,* Empirical Software Engineering **21** (2016).

[104] C. A. Hammerschmidt, S. Garcia, S. Verwer, and R. State, *Reliable machine learning for networking: Key issues and approaches,* in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)* (IEEE, 2017).

[105] G. Pellegrino, Q. Lin, C. A. Hammerschmidt, and S. Verwer, *Learning behavioral fingerprints from netflows using timed automata,* in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)* (IEEE, 2017).

[106] D. Chivilikhin and V. Ulyantsev, *Inferring automata-based programs from specification with mutation-based ant colony optimization,* in *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation* (Association for Computing Machinery, 2014).

[107] M. F. Schilling, *Multivariate two-sample tests based on nearest neighbors,* Journal of the American Statistical Association **81** (1986).

[108] S. Verwer, M. de Weerdt, and C. Witteveen, *A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data,* in *Grammatical Inference: Theoretical Results and Applications* (International Conference on Grammatical Inference (ICGI), 2010).

[109] S. Verwer, R. Eyraud, and C. De La Higuera, *Pautomac: a probabilistic automata and hidden markov models learning competition,* Machine learning (2014).

[110] A. K. Jain, M. N. Murty, and P. J. Flynn, *Data clustering: A review,* ACM Computing Surveys (CSUR) **31** (1999).

[111] G. Pellegrino, *RAI Experiments,* https://github.com/ghibbster/RAI_experiments.git (2018), last accessed: 2023-10-24.

[112] Stratoshpere Lab, *CTU-malware-capture-botnet-25-1,* https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-25-1/2013-11-06_capture-win6.binetflow.labeled.sorted (2013), last accessed: 2023-10-12.

[113] S. García, V. Uhlíř, and M. Rehak, *Identifying and modeling botnet c&c behaviors,* in *Proceedings of the 1st International Workshop on Agents and CyberSecurity*, ACySE '14 (Association for Computing Machinery, 2014).

[114] TU Delft Weather, *Weatherstations rotterdam,* http://weather.tudelft.nl/csv/ (2012), last accessed: 2023-10-12.

[115] G. E. P. Box and G. M. Jenkins, *Some recent advances in forecasting and control,* Journal of the Royal Statistical Society. Series C (Applied Statistics) **17** (1968).

[116] G. E. P. Box, G. M. Jenkins, and G. M. Reinsel, *Time Series Analysis: Forecasting and Control* (Wiley, 1994).

[117] L. Rabiner and B. Juang, *An introduction to hidden markov models,* IEEE ASSP Magazine **3** (1986).

[118] S. Verwer and C. A. Hammerschmidt, *Flexfringe: A passive automaton learning package,* in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (IEEE, 2017).

[119] M. Hulden, *Treba learning tool,* https://code.google.com/archive/p/treba/ (2013), last accessed: 2023-10-12.

[120] C. Hammerschmidt, S. Marchal, R. State, G. Pellegrino, and S. Verwer, *Efficient learning of communication profiles from ip flow records,* in *2016 IEEE 41st Conference on Local Computer Networks (LCN)* (IEEE, 2016).

[121] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes, *Review: A survey of network flow applications,* Journal of Network and Computer Applications **36** (2013).

[122] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)* (Springer-Verlag New York, Inc., 2006).

[123] S. Boucheron, G. Lugosi, and P. Massart, *Concentration Inequalities: A Nonasymptotic Theory of Independence* (Oxford University Press, 2013).

[124] S. Garcia, M. Grill, J. Stiborek, and A. Zunino, *An empirical comparison of botnet detection methods,* Computers & Security **45** (2014).

[125] C. de la Higuera, *A bibliographical study of grammatical inference,* Pattern Recognition **38** (2005).

[126] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont, *A framework for the competitive evaluation of model inference techniques,* in *Proceedings of the First International Workshop on Model Inference In Testing* (Association for Computing Machinery, 2010).

[127] G. Gu, R. Perdisci, J. Zhang, W. Lee, *et al.*, *Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection,* in *Proceedings of the 17th USENIX Security Symposium* (USENIX Association, 2008).

[128] Z. Zhu, G. Lu, Y. Chen, Z. J. Fu, P. Roberts, and K. Han, *Botnet research survey,* in *2008 32nd Annual IEEE International Computer Software and Applications Conference* (IEEE, 2008).

[129] M. Feily, A. Shahrestani, and S. Ramadass, *A survey of botnet and botnet detection,* in *2009 Third International Conference on Emerging Security Information, Systems and Technologies* (IEEE, 2009).

[130] Y. Tang and S. Chen, *Defending against internet worms: A signature-based approach,* in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 2 (IEEE, 2005).

[131] J. R. Binkley and S. Singh, *An algorithm for anomaly-based botnet detection,* in *2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI 06)* (USENIX Association, 2006).

[132] H. Choi and H. Lee, *Identifying botnets by capturing group activities in DNS traffic,* Computer Networks **56** (2012).

[133] O. Pomorova, O. Savenko, and S. Lysenko, *A technique for the botnet detection based on DNS - traffic analysis,* in *Computer Networks: 22nd International Conference, CN 2015, Brunów, Poland, June 16-19, 2015. Proceedings*, Vol. 522 (Springer International Publishing, 2015).

[134] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, *Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware,* IEEE international conference on Dependable Systems and Networks with FTCS and DCC (DSN) (2008).

[135] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, *Learning and classification of malware behavior,* in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Springer, 2008).

[136] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, *A survey on automated dynamic malware-analysis techniques and tools,* ACM Computing Surveys **44** (2012).

[137] G. Jacob, H. Debar, and E. Filiol, *Behavioral detection of malware: from a survey towards an established taxonomy,* Journal in computer Virology **4** (2008).

[138] C. Y. Cho, D. Babić, E. C. R. Shin, and D. Song, *Inference and analysis of formal models of botnet command and control protocols,* in *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Association for Computing Machinery, 2010).

[139] W. T. Strayer, D. Lapsely, R. Walsh, and C. Livadas, *Botnet detection based on network behavior,* in *Botnet Detection* (Springer, 2008).

[140] M. Jaber, R. G. Cascella, and C. Barakat, *Can we trust the inter-packet time for traffic classification?* in *2011 IEEE International Conference on Communications (ICC)* (IEEE, 2011).

[141] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel, *Disclosure: Detecting botnet command and control servers through large-scale netflow analysis,* in *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12 (Association for Computing Machinery, 2012).

[142] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, *Prospex: Protocol specification extraction,* in *2009 30th IEEE Symposium on Security and Privacy* (IEEE, 2009).

[143] C. Goutte, P. Toft, E. Rostrup, F. Å. Nielsen, and L. K. Hansen, *On clustering fMRI time series,* NeuroImage **9** (1999).

[144] A. Khalimonenko, O. Kupreev, and K. Ilganaev, *DDoS attacks in Q3 2017,* https://securelist.com/ddos-attacks-in-q3-2017 (2017), last accessed: 2023-10-11.

[145] P. Collinson, *Alleged mastermind behind bank cyber-attacks extradited to UK,* https://www.theguardian.com/uk-news/2017/aug/30/alleged-mastermind-daniel-kaye-lloyds-bank-cyber-attacks-extradited-uk (2017), last accessed: 2023-10-11.

[146] H. Tiirmaa-Klaar, J. Gassen, E. Gerhards-Padilla, and P. Martini, *Botnets: How to fight the ever-growing threat on a technical level,* Botnets (2013).

[147] G. Gu, J. Zhang, and W. Lee, *BotSniffer: Detecting botnet command and control channels in network traffic,* in *Proceedings of the Network and Distributed System Security Symposium* (The Internet Society, 2008).

[148] M. Abu Rajab, J. Zarfoss, F. Monrose, and A. Terzis, *A multifaceted approach to understanding the botnet phenomenon,* in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC 2006 (Association for Computing Machinery, 2006).

[149] P. Garcia Teodoro, J. Diaz Verdejo, G. Macia Fernandez, and E. Vazquez, *Anomaly-based network intrusion detection: Techniques, systems and challenges,* Computers & Security **28** (2009).

[150] W. C. Lin, S. W. Ke, and C. F. Tsai, *CANN: An intrusion detection system based on combining cluster centers and nearest neighbors,* Knowledge-Based Systems **78** (2015).

[151] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, *Bothunter: Detecting malware infection through ids-driven dialog correlation,* in *USENIX Security Symposium* (USENIX Association, 2007).

[152] S. Garcia, A. Zunino, and M. Campo, *Botnet Behavior Detection using Network Synchronism,* in *Privacy, Intrusion Detection and Response: Technologies for Protecting Networks* (Information Science Reference, 2012).

[153] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, *A view on current malware behaviors,* in *Proceedings of the 2nd USENIX Conference on Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09 (USENIX Association, 2009).

[154] G. E. P. Box and G. Jenkins, *Time Series Analysis, Forecasting and Control* (Holden-Day, Incorporated, 1976).

[155] Stratoshpere Lab, *CTU-malware-capture-botnet-260-1,* https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-260-1 (2017), last accessed: 2023-10-14.

[156] Stratoshpere Lab, *CTU-malware-capture-botnet-261-1,* https://mcfp.felk.cvut.cz/publicDatasets/CTU-Malware-Capture-Botnet-261-1 (2017), last accessed: 2023-10-14.

[157] Microsoft, *Microsoft security bulletin MS03-026,* https://technet.microsoft.com/en-gb/library/security/ms03-026 (2003), last accessed: 2023-10-14.

[158] V. M. Alvarez, *Welcome to YARA's documentation!* https://yara.readthedocs.io/en/v4.0.5/index.html (2021), last accessed: 2023-10-14.

ACKNOWLEDGEMENTS

As you probably understand from my Curriculum Vitae, my professional life wasn't a straight line from one milestone to the next. It was and is still full of overlapping experiences and sudden yet impactful decisions. Along this journey, there was a reasonably long-lasting moment where I was close to giving up on my PhD thesis and forgetting to defend it. Those feelings were caused by a mixture of concurrent causes that have nothing to do with the working environment. Indeed, if I really succeeded in defending my PhD thesis, it is only thanks to a couple of persons I want to immediately mention here.

The first one is my supervisor and promoter, Sicco Verwer. He first believed in me by accepting my PhD application. Sicco has mentored me from my first day in the Netherlands up to the last one. His ability to immediately spot flaws in my ideas and, at the same time, propose fruitful solutions is still unmatched in my professional life. Furthermore, he opened the doors of his house to me and my family and let me discover his wonderful wife and children. Sicco pushed me to pick up my thesis work despite having different thoughts. I'll never have enough words to express my gratitude for what he has done for me.

The second person I must thank is my other promotor Jan van den Berg. Jan helped me in uncountable ways to improve as an aspiring researcher with his regular feedback on my ongoing work and his contagious motivation. He also significantly contributed to improving this thesis with his punctual comments. Thanks once more, Jan!

Among the great people I have encountered in my experience as a PhD student, I want to mention Sebastian Garcia and Veronica Valeros. Sebastian and Veronica hosted me at the Stratosphere Lab of the Czech Technological University in Prague. That was a fantastic period where I learned much about intrusion detection systems and network data analysis. After so many years, I consider them both true friends.

Frits Vaandrager welcomed me at Radboud University when I started my PhD journey. I will always remember the Friday meetings at the social space of the Department of Software Science. Since I moved to Delft, we only met a few times, but on all those occasions, Frits gave me precise and effective suggestions for improving my work and thesis. His last round of feedback before the publication significantly improved my background chapter (and prevented me from writing a considerable amount of inaccuracies!).

I want to thank all the members of my PhD committee for their time spent on my thesis. I have already mentioned a few of them in this chapter. I am honoured by your attendance at such an important event in my life.

Two more people helped me a lot to improve the quality of this thesis. I want to mention them as a form of gratitude: Alex Mandel, who reviewed a few chapters of my thesis as a technical editor, and my wife, Francesca Avitabile, who designed the cover for my thesis and the invitation to the defense ceremony.

I enjoyed working with a few people on similar topics during my PhD years. I want to thank them because they made my days happier, and I learned a lot from their work and experience. They are: Qin Lin, Christian Hammersmith, Rick Smetser, and Paul Fiterau-Brostean. Hope to meet you again, one day or another!

Speaking about colleagues and coworkers, I want to thank Sandra Wolff for being Sandra Wolff! Always kind and always willing to help. I also want to mention a few brilliant people from the former department at the TU Delft tower: Norbert Blenn, Mark Luchs, Vincent Ghiette, Zhijie Ren, Gamze Tillem, Chibuike Ugwuoke, and Majit Nateghizad. They heavily contributed to building a fun and positive working environment.

During my PhD journey, especially at the beginning, it helped me a lot to rely on the support of a handful of friends I met in the Netherlands: Rocco Gaudenzi, Antonio Sanna, Paolo De Martino, and Libera Amenta. They proved their friendship on many occasions and were present in those rare moments of loneliness that may happen when you leave your nest for the first time.

Clearly, all that couldn't happen without the support of my broad and chaotic family, composed of one mother, two fathers, three brothers, and one sister, all scattered across Italy. All of them, starting from my mother, Pino, my siblings, and including my last brother Fabio, never stopped believing in me. I tried not to betray their blind trust and that is one of the reasons why I did not give up with my PhD.

In such a long time, I lost a few persons I wanted to have at my defence ceremony. Above all, my biological father, Giuseppe. We never really got along with each other until we both were alive. Now that he has passed away, I regret all those missed occasions to (re)build relationship. I feel the gap he left in my life.

Last but not least, I want to thank again my wife, Francesca. We met for the first time when we were kids; we were desk pals at primaries, and after so many years, we are not bored seeing each other's faces yet. That is likely primary evidence of love. Crossing Francesca's way has been the best event in my life. Thank you for every word, every call, every exam you helped me to prepare, every good moment, every bad moment, and thanks for the wonderful daughter you gave me. I want the last word of this chapter to be just our daughter's name: Giulia.

Nino Pellegrino
October 2023
Naples, Italy

CURRICULUM VITÆ

Gaetano PELLEGRINO

Gaetano Pellegrino was born on May 19th 1984, in Naples, Italy.

Before graduating, he worked as a software developer in charge of designing, coding, and testing ETL procedures operating on the traffic data of a global telecommunications provider.

He earned his Bachelor's degree in Computer Science at the University of Naples Federico II in 2009 with a thesis on model checking and model verification. His thesis is about developing a model checker based on Alternating-time Temporal logic to validate security policies for large web portals.

He earned his Master's in Computer Science cum laude at the University of Naples Federico II in 2013. His thesis is on applying machine learning techniques to the bioinformatics problem of finding potential correlations between the behaviour of Fe36 protein and the degradation in performance of the anti-DNA-degeneration system proper of the human cells. DNA degeneration is a characteristic of many neurodegenerative illnesses, such as Alzheimer's.

Fascinated about machine learning and eager to work on a different application domain, after graduating, he applied for a PhD position for the LEMMA (Learning Extended State Machines for Malware Analysis) research project at the Radboud University in Nijmegen, the Netherlands. Once obtained the position, he moved to the Netherlands, where he happily lived for five years. Starting in Nijmegen, his PhD journey continued in Delft, the Netherlands, where he followed his supervisor, Dr. Ir. Sicco Verwer, after six months.

In 2018, he had to return to his country without defending his thesis. Gaetano started working as a malware analyst and reverse engineer, one of his passions, in a consultancy role for several corporations in the banking, insurance, and telecommunication businesses. He earned a SANS GIAC Reverse Engineering Malware (GREM) certification and a SANS GIAC Network Forensics Analyst (GNFA) certification during this period.

In 2021, he accepted a position as Senior Threat Researcher at Infoblox, a US-based company, attracted by the opportunity to return to research activities. During this period, he researched threats in DNS data, by focising on homoglyph domains, sound-squatting domains, and automatic domain reputation scoring, topics.

In 2022, he accepted a position as Senior Security Researcher II at ZScaler, a US-based company, to work on malware-related research topics. He joined the Advanced Persistent Threats research group of the ThreatLabZ laboratory, where he researches means to track state-sponsored threat groups and proactively detect targeted attacks in various data sources including web proxy telemetry.

Gaetano loves practicing futsal and swimming. Furthermore, he is passionate about history and deeply interested in the Middle Ages and the Years of Lead. In 2022, he achieved his highest life achievement because he became the father of a wonderful daughter.

EDUCATION

2003–2009	Bachelor in Computer Science University of Naples Federico II *Thesis:* GSOLVER, a Framework for the Solution of Games on Graphs *Promotor:* Dr. Marco Faella
2009–2013	Master *cum laude* in Computer Science University of Naples Federico II *Thesis:* The Protein Fe65, Searching for New Features Using Machine Learning Techniques *Promotor:* Dr. Anna Corazza *Promotor:* Prof. dr. Mario Nicodemi

WORK EXPERIENCE

2023–Pres.	Senior Threat Researcher II ZScaler *Location:* Remote
2021–2022	Senior Threat Researcher Infoblox *Location:* Remote
2018–2021	Associate Manager Accenture Security *Location:* Milan & Naples, Italy
2013–2018	PhD. Candidate Delft University of Technology & Radboud University in Nijmegen *Location:* Delft & Nijmegen, The Netherlands
2007–2009	Junior Programmer Accenture Technology *Location:* Naples, Italy

List of Publications

5. **G. Pellegrino**, C. Hammerschmidt, Q. Lin, S. Verwer, *Learning Deterministic Finite Automata from Infinite Alphabets*, International Conference on Grammar Inference (2017).

4. T. Veugen, J. Doumen, Z. Erkin, **G. Pellegrino**, S. Verwer, J. Weber, *Improved privacy of dynamic group services*, EURASIP Journal on Information Security (2017).

3. **G. Pellegrino**, Q. Lin, C. Hammerschmidt, S. Verwer, *Learning Behavioral Fingerprints From Netflows Using Timed Automata*, Symposium on Integrated Network and Service Management (2017).

2. Q. Lin, C. Hammerschmidt, **G. Pellegrino**, S. Verwer, *Short-termTimeSeriesForecastingwithRegression Automata*, Workshop on Mining and Learning from Time Series (2016).

1. C. Hammerschmidt, S. Marchal, R. State, **G. Pellegrino**, S. Verwer, *Efficient Learning of Communication Profiles from IP Flow Records*, 41st Conference on Local Computer Networks (2016).

APPENDIX A

Table 1: Distribution of labels on the CTU-13 dataset. Each scenario is composed of one or more infected computers and at least three normal computers.

Scenario	Type	IP	Label	#flows	Scenario	Type	IP	Label	#flows
1	Malware	147.32.84.165	Botnet	40,961	9	Malware	147.32.84.206	Botnet	18,553
1	Normal	147.32.84.170	Normal-V42-Stribrek	18,438	9	Malware	147.32.84.207	Botnet	15,999
1	Normal	147.32.84.164	Normal-V42-Grill	7,654	9	Malware	147.32.84.208	Botnet	17,909
1	Normal	147.32.84.134	Normal-V42-Jist	3,808	9	Malware	147.32.84.209	Botnet	16,376
2	Malware	147.32.84.165	Botnet	20,941	9	Normal	147.32.84.170	Normal-V42-Stribrek	15,806
2	Normal	147.32.84.170	Normal-V42-Stribrek	8,960	9	Normal	147.32.84.134	Normal-V42-Jist	9,419
2	Normal	147.32.84.164	Normal-V42-Grill	25	9	Normal	147.32.84.164	Normal-V42-Grill	4,432
3	Malware	147.32.84.165	Botnet	53,518	10	Malware	147.32.84.165	Botnet	9,579
3	Normal	147.32.84.170	Normal-V42-Stribrek	217,614	10	Malware	147.32.84.191	Botnet	10,454
3	Normal	147.32.84.134	Normal-V42-Jist	1,934	10	Malware	147.32.84.192	Botnet	10,397
3	Normal	147.32.84.164	Normal-V42-Grill	9,160	10	Malware	147.32.84.193	Botnet	10,009
4	Malware	147.32.84.165	Botnet	2, 580	10	Malware	147.32.84.204	Botnet	11,159
4	Normal	147.32.84.170	Normal-V42-Stribrek	12,133	10	Malware	147.32.84.205	Botnet	11,874
4	Normal	147.32.84.134	Normal-V42-Jist	10,382	10	Malware	147.32.84.206	Botnet	11,287
4	Normal	147.32.84.164	Normal-V42-Grill	2,474	10	Malware	147.32.84.207	Botnet	10,581
5	Malware	147.32.84.165	Botnet	1,802	10	Malware	147.32.84.208	Botnet	11,118
5	Normal	147.32.84.170	Normal-V42-Stribrek	3,620	10	Malware	147.32.84.209	Botnet	9,894
5	Normal	147.32.84.134	Normal-V42-Jist	2,214	10	Normal	147.32.84.170	Normal-V42-Stribrek	10,216
5	Normal	147.32.84.164	Normal-V42-Grill	3,444	10	Normal	147.32.84.134	Normal-V42-Jist	1,091
6	Malware	147.32.84.165	Botnet	9,260	10	Normal	147.32.84.164	Normal-V42-Grill	3,728
6	Normal	147.32.84.170	Normal-V42-Stribrek	10,976	11	Malware	147.32.84.165	Botnet	4,151
6	Normal	147.32.84.134	Normal-V42-Jist	1,364	11	Malware	147.32.84.191	Botnet	4,006
6	Normal	147.32.84.164	Normal-V42-Grill	2,490	11	Malware	147.32.84.192	Botnet	7
7	Malware	147.32.84.165	Botnet	126	11	Normal	147.32.84.170	Normal-V42-Stribrek	581
7	Normal	147.32.84.170	Normal-V42-Stribrek	1,614	11	Normal	147.32.84.134	Normal-V42-Jist	11
7	Normal	147.32.84.134	Normal-V42-Jist	584	11	Normal	147.32.84.164	Normal-V42-Grill	2,113
7	Normal	147.32.84.164	Normal-V42-Grill	1,040	12	Malware	147.32.84.165	Botnet	807
8	Malware	147.32.84.165	Botnet	12,254	12	Malware	147.32.84.191	Botnet	766
8	Normal	147.32.84.170	Normal-V42-Stribrek	97,176	12	Malware	147.32.84.192	Botnet	570
8	Normal	147.32.84.134	Normal-V42-Jist	10,926	12	Normal	147.32.84.170	Normal-V42-Stribrek	4,359
8	Normal	147.32.84.164	Normal-V42-Grill	36,328	12	Normal	147.32.84.134	Normal-V42-Jist	2,145
9	Malware	147.32.84.165	Botnet	22,792	12	Normal	147.32.84.164	Normal-V42-Grill	1,075
9	Malware	147.32.84.191	Botnet	18,774	13	Malware	147.32.84.165	Botnet	40,003
9	Malware	147.32.84.192	Botnet	20,305	13	Normal	147.32.84.170	Normal-V42-Stribrek	26,846
9	Malware	147.32.84.193	Botnet	17,961	13	Normal	147.32.84.134	Normal-V42-Jist	948
9	Malware	147.32.84.204	Botnet	18,783	13	Normal	147.32.84.164	Normal-V42-Grill	3,539
9	Malware	147.32.84.205	Botnet	17,535					

APPENDIX B

Table 2: PAutomaC competition problems meta-informations. Each problem, identified by the corresponding id, is a probabilistic stateful model.

ID	START STATES	END STATES	STATES	DETERMINISTIC?	SYMBOL SPARSITY	TRANSITION SPARSITY	ALPHABET SIZE	TYPE
1	5	20	63	NO	0.3274	0.0872	8	hmm
2	1	20	63	NO	0.3280	0.0166	18	hmm
3	1	19	25	NO	0.7900	0.0790	4	pfa
4	1	5	12	NO	0.4375	0.1508	4	pfa
5	1	16	56	NO	0.2946	0.0217	6	hmm
6	1	9	19	YES	0.4825	0.0526	6	dpfa
7	1	2	12	YES	0.2372	0.0833	13	dpfa
8	3	17	49	NO	0.3622	0.0645	8	pfa
9	1	27	71	YES	0.3873	0.0141	4	dpfa
10	1	30	49	NO	0.6327	0.0221	11	pfa
11	1	23	47	YES	0.4947	0.0213	20	dpfa
12	1	4	12	NO	0.3526	0.1116	13	pfa
13	1	43	63	YES	0.6905	0.0159	4	dpfa
14	1	7	15	NO	0.4944	0.0800	12	hmm
15	1	10	26	NO	0.4121	0.0672	14	pfa
16	1	30	49	YES	0.6184	0.0204	10	dpfa
17	3	4	22	NO	0.2168	0.1738	13	pfa
18	1	5	25	YES	0.2260	0.0400	20	dpfa
19	2	22	68	NO	0.3256	0.0350	7	hmm
20	1	4	11	NO	0.3939	0.1570	18	hmm
21	2	14	56	NO	0.2531	0.0497	23	hmm
22	3	13	55	NO	0.0575	0.2411	21	pfa
23	3	12	33	NO	0.3810	0.1148	7	hmm
24	1	2	6	YES	0.5000	0.1666	5	dpfa
25	1	23	40	NO	0.5775	0.0456	10	hmm
26	1	42	73	YES	0.5868	0.0137	6	dpfa
27	1	12	19	YES	0.6378	0.0526	17	dpfa
28	2	17	23	NO	0.7464	0.1134	6	hmm
29	1	13	36	NO	0.3750	0.0384	6	pfa
30	1	5	9	NO	0.6555	0.1751	10	pfa
31	2	4	12	NO	0.3833	0.1992	5	pfa
32	1	33	43	YES	0.7733	0.0233	4	dpfa
33	1	7	13	NO	0.5949	0.1183	15	hmm
34	1	23	64	NO	0.3705	0.0281	21	pfa
35	1	16	47	YES	0.3553	0.0213	20	dpfa
36	4	34	54	NO	0.6317	0.0748	9	hmm
37	12	35	69	NO	0.5217	0.1825	8	pfa
38	2	11	14	NO	0.7857	0.1939	10	hmm
39	1	2	6	NO	0.4167	0.1810	14	pfa
40	1	42	65	YES	0.6473	0.0154	14	dpfa
41	7	37	54	NO	0.6931	0.1430	7	hmm
42	1	3	6	YES	0.5185	0.1667	9	dpfa
43	10	39	67	NO	0.5970	0.1641	5	pfa
44	4	46	73	NO	0.6333	0.0561	13	hmm
45	1	11	14	NO	0.8008	0.0867	19	hmm
46	1	9	19	NO	0.4851	0.0973	23	pfa
47	1	18	61	YES	0.3027	0.0164	15	dpfa
48	1	11	16	YES	0.6957	0.0625	23	dpfa