Chapter 1

Automated Orchestration of Security Chains Driven by Process Learning

Nicolas Schnepf,[1*] Rémi Badonnel,[2] Abdelkader Lahmadi,[2] and Stephan Merz[2]

[1] *Department of Computer Science, Aalborg University, 9220, Aalborg, Denmark*
[2] *Université de Lorraine, CNRS, Loria, Inria, 54000, Nancy, France*

*Corresponding Author

Abstract:

Connected devices, such as smartphones and tablets, are exposed to a large variety of attacks. Their protection is often challenged by their resource constraints in terms of CPU, memory and energy. Security chains, composed of security functions such as firewalls, intrusion detection systems and data leakage prevention mechanisms, offer new perspectives to protect these devices using software-defined networking and network function virtualization. However, the complexity and dynamics of these chains require new automation techniques to orchestrate them. This chapter describes an automated orchestration methodology for security chains in order to secure connected devices and their applications. This methodology exploits process learning to establish behavioral models and infer security

constraints represented as logical predicates. It then generates and merges a set of chains of security functions on the basis of these predicates. These chains are finally compiled into low-level configuration rules and deployed into the network, optimizing for the underlying topology. The benefits and limits of such a methodology combining machine learning and verification techniques are evaluated by a set of experimental results.[1]

Keywords: Security Management, Software-Defined Networking, Chain Synthesis, Process Learning

1.1. Introduction

The relentless growth in the number of connected smart devices such as smartphones and tablets has attracted the attention of malicious actors who exploit these devices as both targets and vectors of attacks against user data and the network infrastructure. For example, four million malicious applications were detected on the Google Play Store in 2019 [22]. Although necessary, preventive screening of applications by the store operators is not sufficient for detecting all malicious applications. Moreover, limited resources in terms of CPU and battery makes it difficult to develop and deploy sophisticated on-device security mechanisms: this certainly applies to IoT applications, but can be true even for smartphones or tablets, depending on the nature of expected processing. Finally, end users may be overwhelmed by the technical details and unaware of unintended functionality that such applications exhibit.

With the development of software-defined networking (SDN), it is attractive to deploy chains of security functions—including firewalls (FW), intrusion detection systems (IDS), deep packet inspection (DPI) or data leakage prevention (DLP) mechanisms—into cloud infrastructures for a network-based protection. SDN relies on de-

[1]Partially supported by the Concordia project that has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement No. 830927.

coupling the network into the data plane, realized by SDN switches, that forward traffic according to configuration rules, and the control plane, commonly realized by a single controller, that reconfigures the switches. The standard OpenFlow protocol may typically support communications between the controller and switches. Although we do not rely on it here, network function virtualization (NFV) provides an elegant abstraction for implementing such services.

However, deploying security chains in practice can be challenging. Their complexity and dynamics is prone to inconsistencies and misconfigurations, resulting in security breaches that could be exploited by attackers. As we discuss in Section 1.2, there has been interesting work on applying formal methods in view of verification or synthesis of chains. However, formal verification techniques tend to exhibit exponential complexity in terms of response time, limiting their applicability in practice, in particular for dynamic deployment in the network. Moreover, mobile applications have heterogeneous networking behavior and vulnerabilities and therefore require tailored security chains for protecting end users and the network infrastructure itself.

This chapter proposes a method for automatically generating security chains for deployment in SDN infrastructures. This method is driven by the security requirements based on the networking behavior of individual applications that we infer using process learning methods. We classify potential attacks using logical predicates and then infer security rules that are grouped into security functions. The resulting application-specific security chains are merged and optimized in view of deploying them in the network.

The remainder of this chapter is organized as follows: Section 1.2 presents related work, Section 1.3 introduces background notions, Section 1.4 gives an overview on the proposed method, whereas the following Sections 1.5–1.8 describe in more detail the different steps for learning networking behavior, generating security chains, formal verification of correctness properties, and optimization. Section 1.9 presents results of performance evaluations and Section 1.10 concludes the chapter and points out future research perspectives.

1.2. Related work

This section is centered on existing work related to chains of security functions and formal verification techniques, in the context of the protection of smart devices and their applications. Different methods have already been designed to mitigate attacks targeting smart devices. These environments are exposed to a large variety of attacks, such as denial of service attacks, port scans, worms and botnets [31]. Android systems are particularly concerned with a growing number of malicious applications, as discussed in [17]. In addition, their resources are often limited, making it impractical to deploy advanced security mechanisms on the devices [31]. The permission system of Android is an important element contributing to security [44]. Permissions grant applications the right for using different resources of the Android device, such as for instance the Internet connection or the cameras. Nevertheless, this system may be the source of misconfigurations [1]. It is possible to monitor the method calls of the applications and compare them to the declared permissions to detect misbehaviors, such as in [4]. However, this approach shows limitations with respect to attacks targeting the network infrastructure, as the Internet permission does not provide a fine granularity. Establishing network profiles of applications has also been explored by several authors. In [44], a security monitoring framework was proposed to combine permissions, user interactions, system calls, and network traffic, whereas the solution developed in [30] permits to learn the communication behaviors of Android applications from their binary files. These approaches are mainly intended for screening new applications rather than protecting end users from malicious behaviors of already installed applications. The lack of reactive methods for protecting devices from installed malicious applications, together with the constrained resources of these environments, goes in favor of exploring learning techniques for detecting specific misbehaviors and developing protective measures that can be outsourced from the devices.

Chains of security functions. The development of software-defined networking (SDN) as well as network function virtualization (NFV) has contributed to the de-

ployment of security chains [18]. In particular, Virtual Network Functions (VNF) can be deployed to enforce security mechanisms. They correspond to network functions that are virtually deployed on commodity hardware [8], and may implement different security functions. These functions can then be chained using the facilities offered by software-defined networks.

There exists a large body of literature addressing the challenge of formally modelling security functions implemented as VNFs. In [16], a generic model is proposed, based on a dedicated language to express security functions. This language relies on pre-conditions and post-conditions regarding the network traffic accepted by a security function. While it provides a very precise and explicit specification of security functions, it suffers from the lack of concrete implementations that would enable the practical deployment of the specified functions. In addition, the high diversity of security functions available on the market makes it difficult to design a generic language without losing semantic properties. In [24], a method is introduced for resolving conflicts that may occur when combining several security functions. While the work has been implemented, it is only focused on the case of firewalls, and does not cover any other security functions. A major issue in the field of security function modelling is therefore to build a model that is general enough to ensure a large coverage, and that can be exploited in practice to support an automated deployment of security functions.

A second challenge concerns more specifically the deployment and configuration of security functions. The use of NFV jointly with the programmability provided by SDN enables a more flexible deployment of security policies. For instance in [3], a framework is described for chaining network and security functions implemented as middleboxes based on a high-level specification of composition. This latter is then translated into low-level rules that are interpreted by SDN infrastructures, in order to deploy security chains. Several research efforts analyze the deployment of security functions as a resource allocation problem: indeed the problem is known to be NP-hard in the general case. In [35], a three-stage formalization is considered: service function chain (SFC) composition, SFC embedding and SFC scheduling. The first stage of the

problem is solved by characterizing the service requests in terms of network functions, and optimally building the service function chains using an integer linear programming (ILP) approach. The allocation takes into account that multiple service function chains may exploit the same virtualized network functions, but also that there may exist dependencies amongst some of the VNFs, requiring to place them in a specific order. In the same manner, the authors of [36] explore automated security function allocation to support reactive security in 5G infrastructures. The proposed framework relies on SDN supervisory control and data acquisition honeypots. It follows the standardization efforts from the IETF SFC working group, and exploits OpenDayLight controllers to configure the infrastructure. It permits a continuous monitoring of industrial networks and a fine-grained analysis of potential attacks that then serves to isolate attackers and evaluate their level of sophistication.

Another important challenge is to exploit security patterns to drive the configuration of security chains. In particular, network security patterns have been introduced for leveraging the best practices from the security experts, and capturing different security constraints that enable the efficient selection of adequate security functions [41]. That paper also introduces a scalable networking and computing resources-aware optimization framework to properly provision different chains based on an open-source cloud environment. For Android environments, the system developed in [25] can be used to analyze the behavior of applications and to build behavioral patterns. These are then exploited to select pre-configured security functions when some deviations from the behavioral patterns are observed. This approach is extended by [26], which integrates in the decision process the permissions initially declared by the applications. However, the security functions are not automatically chained in these scenarios.

Formal verification of networking policies. Formal verification techniques are a key enabler for automating the orchestration of security chains. Model checking [15] designates a collection of techniques for evaluating if a property (typically expressed as a formula of temporal logic) is true in a structure, such as a transition

system. These techniques were originally applied to the verification of concurrent and distributed systems, and their main limitation is the exponential growth of the number of reachable states in terms of the number of system components. Satisfiability modulo theory (SMT) [11] is also relevant to our work. SMT extends the satisfiability problem of propositional logic (SAT) by considering decidable theories such as fragments of arithmetic, the theory of binary words or strings. Again, SMT solving is at least NP-hard, but works surprisingly well in many practical cases.

These techniques have received much interest in recent years in the context of software-defined networking, in order to check the consistency of network policies before their deployment. The programmability of networks may introduce misconfigurations, and even configuration vulnerabilities that can then be exploited by attackers. For instance, techniques developed in [7] target the verification of the control plane of network infrastructures. Considering a collection of router configurations and a high-level specification of the network behaviors, the approach checks that these configurations correctly enforce the specification for all possible network behaviors. Alternatively, the solution can be exploited to synthesize correct configurations from the high-level specification, to be implemented by network routers. In a similar manner, the Vericon framework [5] is designed to verify that a SDN program (north-bound interface) is correct for all admissible topologies and for all possible sequences of network events. It exploits first-order logic to specify admissible network topologies and desired network-wide invariants, that are then implemented using deductive verification with the Z3 prover as an automatic backend. However, the approach does not take into account temporal logics, which may restrict its overall coverage for preventing some security attacks, such as Denial of Service (DoS) attacks.

Another important challenge is to verify the correctness of updates that are applied to the network configuration at runtime. For instance, [20] proposes a solution based on model checking for the verification of network updates. In this approach, each state of the built automaton corresponds to a state of the network, transitions capture the events affecting the network, such as the sending of packets or the deployment of new

rules. This approach mainly focuses on rule updates and is designed to verify that the configuration of the network remains correct after updates are applied. Due to the considered level of granularity, it seems difficult to be applied in a fully dynamic context, and provides better performance with an offline usage. An alternative solution [28] aims at verifying network-wide invariants that are checked at runtime. The objective is to tame the complexity of the models by considering incremental rather than overall verification. The solution targets the verification of new rules, with respect to the remainder of the network policy. However, it requires the specification of invariants that are defined manually by network experts.

Formal verification has been largely used for checking firewall policies. For instance, SMT solving methods have been used to detect anomalies in large and distributed firewall policies [2]. The insertion or modification of filtering rules may impact the security of the infrastructure and its services. The solution aims at detecting conflicts and redundancies that may occur amongst firewall rules. Two rules are in conflict when they correspond to contradictory decisions, while two rules are redundant when they partially or fully overlap. The verification is performed both in an intra-firewall manner (concerning the rules of the same firewall), and in an inter-firewall manner (concerning the rules that are distributed over several firewalls). Process algebra has also been exploited [27] to support formal verification for SDN-based firewalls. The verification is performed at each update of the network configuration.

1.3. Background

The method that we propose requires some background elements with respect to the flow-based detection of attacks and to programming SDN controllers.

Flow-based detection of attacks. According to RFC 5101 [14], network flows can be defined as collections of IP packets observed at a certain point in the network during a certain time interval. They are generally described by different attributes such

as source and destination IP addresses and port numbers (*srcaddr*, *dstaddr*, *srcport*, and *dstport*), their network protocol (*protocol*) and the numbers of packets or bytes they contain (*packets* and *bytes*). We assume that flows are collected on-device [32], and extended with a timestamp (*timestamp*) and the name of the application that produced them (*appname*). Although the network flows do not represent the payload transmitted during a communication, their analysis can indicate certain kinds of security attacks [43]. Combining the *appname* attribute with the permission system of Android enables furthermore gaining some insight into the kind of data that may be transmitted in a flow.

Denial of service (DoS) attacks target a victim in order to prevent it from providing a service [23]. We consider DoS attacks that can be observed from a networking point of view in that they produce abnormal quantities of traffic from or to a certain equipment. For example, in a SYN flood attack a large number of SYN packets are sent to a host in order to overload the TCP stack with connections that will never be closed.

In port scanning attacks, an application initiates connections with multiple port numbers in order to detect open ports. For example, the port scanner `nmap` available on standard Linux platforms gives rise to characteristic patterns in network flows.

A worm is a program that can execute independently while consuming the resources of its host and that can replicate a fully executable version of itself to other devices [33]. Worms replicate by exploiting vulnerabilities of applications and operating systems, or by methods of social engineering. We consider worms that scan certain ports on devices.

A potentially malicious bot is a program installed on a system in order to execute tasks, typically under the control of a remote administrator, called bot master [6]. The detection of botnets has been extensively studied. In particular, certain botnets communicate using HTTP requests that are hard to identify from a networking point of view, and some are based on a peer-to-peer architecture in order to transmit messages of the bot master. We consider botnets that can be detected based on the large amount of traffic that they exchange with their controller or by the use of network protocols

that are abnormal in a certain context.

The objective here is to detect such attacks by profiling the behavior of an application, based on methods of process learning. Modeling the interactions of an application as a Markov automaton, we leverage methods designed to infer the automaton structure such as the K-tail algorithm [12] or its extensions Synoptic [9] or Invarimint [10]. These methods sometimes result in overly complicated models, and we introduce techniques for reducing this complexity in order to make them applicable for dynamically orchestrating security chains for smart devices.

Programming SDN controllers. Whereas SDN controllers typically use the OpenFlow protocol to communicate with programmable switches, several higher-level languages have been designed for programming them. Our method is based on the Pyretic language [21], part of the Frenetic [19] family of programming languages developed by Foster, Rexford et al. This programming language, implemented in Python, describes the behavior of the data plane for any kind of traffic accepted by the network. Pyretic provides some basic policies as well as operators for combining policies. The basic policies include:

- *identity* to forward all incoming packets,
- *drop* to remove all incoming packets,
- $match(x_1 = y_1, \ldots, x_n = y_n)$ to forward packets whose header fields x_i equal y_i,
- $modify(x_1 = y_1, \ldots, x_n = y_n)$ to forward all packets and changes the header fields x_i to y_i,
- *query* to send packets to the controller for deeper analysis,
- $countPackets(x_1 = y_1, \ldots, x_n = y_n)$ to count the number of packets whose header fields x_i contain the values y_i,
- $limitFilters(k, x_1 = y_1, \ldots, x_n = y_n)$ to forward at most k packets whose header fields x_i contain the values y_i,
- $regexpQuery(pattern)$ to forward packets whose payload matches the given regular expression.

Operators for combining policies include sequential composition, parallel composition, and complement. The sequential composition $p_1 \gg p_2$ forwards all packets accepted by both policies p_1 and p_2 (where p_2 receives packets accepted and potentially modified by p_1). The parallel composition $p_1 + p_2$ forwards all packets accepted by p_1 or by p_2, whereas the complement $\sim p_1$ forwards all packets rejected by p_1 and vice versa.

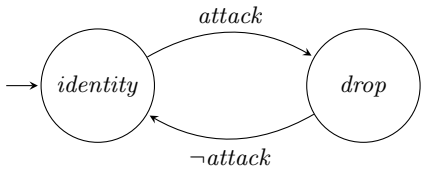

Figure 1.1: Example of a Kinetic control plane automaton.

The Kinetic [29] extension of the Pyretic language enables the verification of the control plane described as a finite state automaton. As a simple example, Fig. 1.1 illustrates an automaton that switches between the identity and the drop policies on the basis of the detection of an attack. The idea is that the traffic is normally forwarded without any further control unless an attack is detected, which would cause the traffic to be dropped. Kinetic users can provide properties expressed in the CTL temporal logic and verify the control plane automaton against this property. However, verification is restricted to the control plane in Kinetic, and properties of the data plane cannot be verified.

1.4. Orchestration of security chains

We now describe a collection of techniques for orchestrating chains of security functions that are deployed in SDN environments. The illustrative use case corresponds to the protection of smart devices with limited CPU and battery capacities such as presented

in [37], in particular Android devices, but the methodology is also applicable to SDN infrastructures in general. We give here an overview of the security chain orchestrator, while the different steps related to the orchestration methodology will be described in more detail in the subsequent sections.

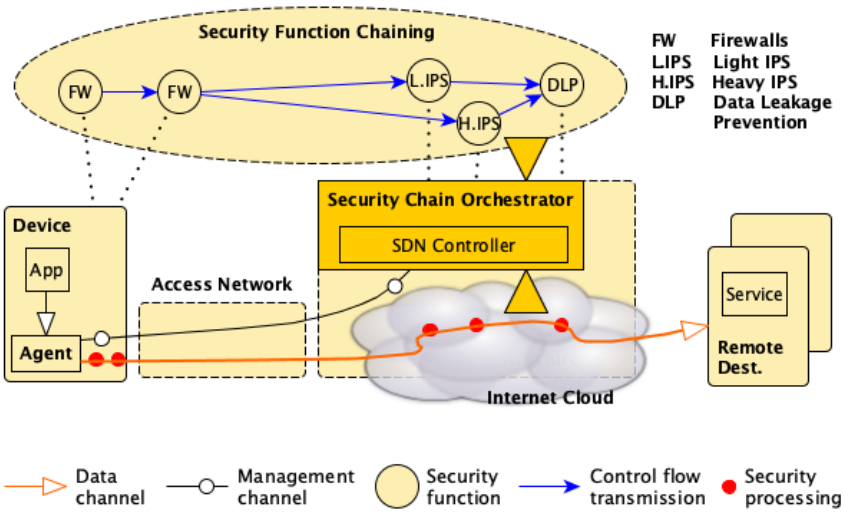

Figure 1.2: Security chain orchestrator integrated into a SDN infrastructure.

A high-level picture of the orchestrator is provided in Figure 1.2. An agent installed on the device shown on the bottom left registers the security requirements of the installed applications. As discussed in Section 1.2, the actual security functions such as firewalls or intrusion detection systems are deployed in a cloud infrastructure, symbolized by red points in the cloud in the bottom part. These security functions are orchestrated by the security orchestrator which exploits different techniques to build, verify and optimize the chains of security functions. These are then compiled into low-level configuration rules and transmitted to the controller in order to be deployed. The purpose of the chains is to filter the traffic between the device and the remote destinations that it contacts, represented on the right. Devices transmit to the orchestrator the list of applications that connect to the network. Security requirements related to an application are inferred based on a model of its networking interactions in terms

of network flows as well as on the permissions requested by the application in its manifest file. While network flows do not represent the data transmitted in messages, the permissions declared by the application are used in order to over-approximate the data that may be exchanged.

The four main problems that we address are the following, and correspond to the different steps of the proposed method (as shown in Fig. 1.3):

1. Build a model of the security requirements of the applications to be protected;
2. Synthesize automatically the chains of security functions;
3. Verify that the generated security chains meet the requirements;
4. Optimize their deployment in order to minimize the impact on the network.

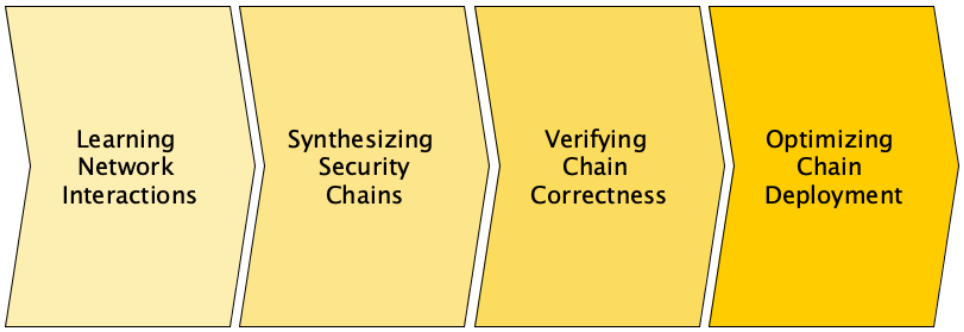

Figure 1.3: Different steps of the proposed methodology.

The first step includes constructing concise and accurate models of the networking behavior of an application; it is addressed by applying process learning techniques. This step results in a finite automaton (more precisely, a Markov chain) that represents the networking interactions of an application based on flow traces collected during its execution. This model is analyzed in order to detect anomalies that may indicate some malicious behavior of an application. These anomalies are represented as predicates that will be used by the subsequent steps for generating abstract representations of chains of security functions that will then be optimized before being compiled into a concrete implementation and deployed.

Concretely, the predicates inferred from the behavior model permit to generate functional representations of single chains (second step). Chains corresponding to individual applications can be combined in order to factor common parts and minimize the overall number of rules to be deployed. They are also formally verified to check their consistency and user-specified correctness properties (third step). Finally, an optimization step computes the optimal placement of security rules according to the topology of the network and criteria specified by the network operator (fourth step).

1.5. Learning network interactions

Process learning techniques are applied for modeling the networking behavior of applications, as presented in more detail in [39]. In preparation to the construction of a security chain, network flows for an application are collected by the Flowoid agent [32] that is deployed on the device, and they are then collected as a dataset. For our experimental evaluation, we consider a pre-existing dataset of flows of multiple Android applications.

These learning techniques are of limited use when applied to strongly heterogeneous datasets. Network flows typically contain many different IP addresses that correspond to a single service provider. The flows collected by Flowoid are therefore enriched by a field representing the owner of an IP address. This piece of information, abbreviated as *orgname*, can be retrieved using the well-known `whois` tool, which also provides the *netname*, i.e. the name of the network in which the IP address is deployed. Usually, the netname is more specific than the orgname, and we decide on which of the two fields to use based on a threshold for the number of occurrences.

Although `whois` is still the most widely used tool for querying the owner of an IP address, it is also quite common that this information is not available or outdated, motivating the interest for possible alternatives. A first good candidate is the RDAP protocol [34] proposed as a successor to `whois`. This protocol is based on HTTPS

and provides its answer in the JSON format. A second possible alternative is the reverse DNS protocol (RDNS) used to retrieve the domain name associated with an IP address. This solution is actually used by most mail servers to filter out IP addresses that do not belong to any domain name, which could also be used in our approach to identify unsafe IP addresses.

After collecting and enriching the flows of an application we use them to build its behavioral model. A representation in the form of a finite automaton with probabilistic transitions appears particularly appropriate, and we examined existing techniques for learning automaton structures such as the K-tail algorithm [12] or its Synoptic [9] extension, as well as Invarimint [10]. These three methods receive a list of the logs of a system and output an automaton describing the behavior that can be derived from the input logs. Both K-tail and Synoptic learn a Markovian automaton whose transitions are labeled by probabilities, the limit of these approaches is nevertheless the high level of complexity of their outputs. In contrast, Invarimint produces a simpler automaton without probabilities that qualitatively describes the behavior observed in the input logs. We found that on our datasets, Synoptic produced overly complicated automata while Invarimint produced simpler automata, but it does not take into account probabilities.

We therefore designed an algorithm that produces a Markov chain (similar to Synoptic) while producing a compact representation (similar to the automata generated by Invarimint). Algorithm 1 represents the automaton using the tables *States* and *Transitions*. It takes as input a list of size N of orgnames, obtained from the flows in the dataset by splitting them into chunks with identical orgname attribute. Automaton states correspond to orgnames, while transitions indicate the probability of succession between orgnames.

The algorithm creates an automaton with as many states as the input contains orgnames, and the weight of a state corresponds to how often it appears. For every pair of successive states, a transition is created and its weight is computed similarly. At the end of the algorithm, transition probabilities are assigned by dividing the

Algorithm 1 Learning a Markov chain.

Input: *flow*, a list of size $N + 1$ of orgnames (or netnames)
$States := \emptyset$
$Transitions := \emptyset$
$orgname := flow[0]$
$States[orgname] := 1$ ▷ Count the occurrences of states and of transitions
for $i \in 1..N$ **do**
  $transition := (orgname, flow[i])$
  $orgname := flow[i]$
  if $orgname \in States$ **then**
    $States[orgname] \mathrel{+}= 1$
  else
    $States[orgname] := 1$
  end if
  if $transition \in Transitions$ **then**
    $Transitions[transition] \mathrel{+}= 1$
  else
    $Transitions[transition] := 1$
  end if
end for
▷ Compute the probability of each transition
for $transition \in Transitions$ **do**
  $Transitions[transition] := Transitions[transition]/States[transition_0]$
end for

weight of a transition by the weight of its source state. The states of the automaton can be enriched in order to express more information contained in the original flows. Concretely we compute the following standard network metrics from the flows directed to addresses corresponding to each state l of the automaton; these will be used in the following for generating chains of security functions:

- $l.ports$: the set of ports appearing in flows for l;
- $l.protocols$: the protocols used;
- $l.count(x)$: the highest number of occurrences of the address or port x;
- $l.avg\_size$: the average number of packets;
- $l.avg\_interval$: the average distance between communications based on timestamps.

Moreover, $bgp\_ranking(ip)$ denotes a metric corresponding to a value of trust of the IP address ip. In practice, this value is obtained by contacting a remote service

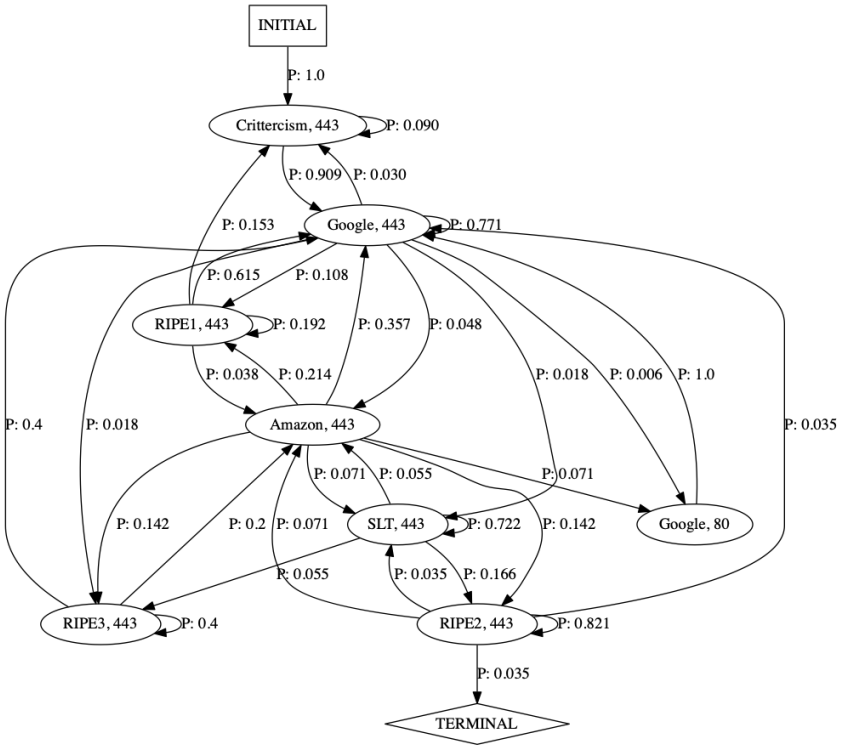

Figure 1.4: Inferred Markov chain of the Pokemon Go Android application.

relying on various data sources to compute the trust ranking of an IP address.

As a concrete example, Fig. 1.4 shows the automaton computed for the dataset corresponding to the Pokemon Go application.[2] Compared to similar automata computed by existing algorithms, our automata have 29.6 states and 141.5 transitions on average against 27.6 states and 142.5 transitions on average for Invarimint. The automata sizes are therefore comparable, but our automata include transition probabilities, and they are much more compact than the automata computed by Synoptic (55 states and 150 transitions on average).

[2]Probabilities have been rounded and may not add up to 1.

1.6. Synthesizing security chains

From the Markov chain representing the network interactions of an application and some thresholds set by the network operator, the next step in our method is to synthesize a high-level representation of a chain of security functions designed to protect the application against the types of attacks mentioned in section 1.3. Indeed, once a user inadvertently installed a malicious application on a smart device it is then necessary to protect the user as well as the network against potential attacks. The Markov model of the behavior of an application helps detect suspicious application behavior and prevent unfortunate consequences for the user or the network.

We use a rule-based approach for generating security chains in order to make the algorithm easy to understand and easy to adapt. We assume the following thresholds corresponding to the metrics introduced previously; concrete values for each of these will be set by network operators.

- *attack_limit*: maximal probability of transitions looping on a single state,
- *min_interval*: minimal interval between flow arrivals,
- *min_size*: minimal number of packets in a flow,
- *ip_limit*: maximal number of occurrences for an IP address,
- *port_limit*: maximal number of occurrences for a port number,
- *port_scan_limit*: maximal number of ports in a flow,
- *unsafe_threshold*: maximal value of *bgp_ranking*.

We also assume given a set $\mathcal{D}_{danger}$ of Android permissions considered as potentially dangerous. Given a Markov chain with states L_{app} and transitions T_{app}, each of the form (l, p, l') for states $l, l' \in L_{app}$ and a probability $p \in [0; 1]$, as well as trace t_{app}, observe that every flow record $f \in t_{app}$ corresponds to precisely one state $l \in L_{app}$, corresponding to $f.orgname$; we denote this state as l_f.

The core of the detection corresponds to an algorithm for classifying destination addresses a appearing in flows of t_{app}. Instead of hardwiring a fixed classification algorithm, we represent each class of attack as a logical predicate and associate with

it a rule that characterizes flows that exhibit the respective attack. In our work, we used the rules shown below; however, they can be modified based on the domain knowledge of the network operator. Use of a declarative programming framework (Prolog in our implementation) helps making these definitions readable and easy to change.

$$
\begin{aligned}
dos(a) \leftarrow\ & \exists f, p:\ f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge p \geq attack\_limit \wedge \\
& l_f.count(a) \geq ip\_limit\ \wedge l_f.avg\_interval \leq min\_interval \wedge \\
& l_f.avg\_size \leq min\_size \\
port\_scan(a) \leftarrow\ & \exists f, p:\ f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge p \geq attack\_limit \wedge \\
& l_f.count(a) \geq ip\_limit \wedge l_f.avg\_interval \leq min\_interval \wedge \\
& l_f.avg\_size \leq min\_size \wedge |\, l_f.ports \,| \geq port\_scan\_limit \\
worm(a, pt) \leftarrow\ & \exists f, p:\ f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge (l_f, p, l_f) \in T_{app} \wedge \\
& p \geq attack\_limit \wedge l_f.count(pt) \geq port\_limit \\
botnet(a, pt) \leftarrow\ & \exists f:\ f \in t_{app} \wedge a = f.dstaddr \wedge \\
& l_f.count(a) \geq ip\_limit \wedge pt = f.dstport \quad \vee \\
& l_f.protocols \cap \{tcp, udp\} \neq \emptyset \wedge l_f.avg\_interval \leq min\_interval \\
unsafe(a) \leftarrow\ & \exists f:\ f \in t_{app} \wedge a = f.dstaddr \wedge bgp\_ranking(a) \geq unsafe\_threshold \\
safe(a) \leftarrow\ & \neg dos(a) \wedge \neg port\_scan(a) \wedge \neg unsafe(a) \wedge \\
& \neg \exists pt : (worm(a, pt) \vee botnet(a, pt)) \\
danger(pm) \leftarrow\ & pm \in P_{f.appname} \cap \mathcal{D}_{danger}
\end{aligned}
$$

Based on these classification rules, we associate elementary security rules with IP addresses that appear in the trace. These rules are then composed in parallel, yielding security functions such as firewalls or intrusion detection systems that are in turn composed in sequence for building chains of security functions. We continue to describe our methodology using declarative rules, and later explain how to translate these into a Pyretic program.

We represent network traffic as a sequence $t \in \mathcal{P}^*$ where $\mathcal{P}$ denotes the set of

network packets. A security function $f : \mathcal{P}^* \to \mathcal{P}^*$ transforms network traffic. For an integer $n \in \mathbb{N}$, the function $cut(t, n)$ returns the prefix of t of length (at most) n. Given a predicate $pred(p)$ on packets, the function $restrict(t, pred)$ returns the subsequence of t of those packets satisfying $pred$.

Given two traces t_1 and t_2, their *merge* $t_1 \oplus t_2$ corresponds to the unique trace formed by the elements of t_1 and t_2 in increasing order of time stamps, with the proviso that whenever t_1 and t_2 contain flows f_1 and f_2 with $f_1.timestamp = f_2.timestamp$, then f_1 appears in $t_1 \oplus t_2$ while f_2 is dropped. Security functions can be composed in sequence ($\circ_{\gg}$) or in parallel ($\circ_+$):

$$(f \circ_{\gg} g)(t) = g(f(t)) \qquad (f \circ_+ g)(t) = f(t) \ \oplus \ g(t)$$

and these operators generalize to n-ary compositions $\bigcirc_{\gg}$ and $\bigcirc_+$.

Elementary security rules make use of the following predicates that can be implemented directly in Pyretic or using VNF rules if we were using Network Function Virtualization:

- $regexp(s, pm)$: true if the string s, representing the payload of the packet, satisfies the regular expression associated with the permission pm;
- $tcp\_check(t)$: true if the traffic t respects the standards of a TCP connection;
- $http\_check(s)$: true if the string s, representing the payload of the packet, is a valid HTTP request;
- $inspect\_payload(s)$: true if the string s, representing the payload of the packet, complies with the underlying deep packet inspection (DPI) policy.

We now define elementary security rules:

$$forward(a,t) = restrict(t,\ \lambda pk : pk.dstaddr = a)$$

$$block(a,pt,t) = restrict(t,\ \lambda pk : pk.dstaddr \neq a \wedge pk.dstport \neq pt)$$

$$limit(a,n,t) = cut(forward(a,t),n)$$

$$filter(a,pm,t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge regexp(pk.payload,pm))$$

$$inspect(a,t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge inspect\_payload(pk.payload))$$

$$tcp(a,pt,t) = \begin{cases} restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt) & \text{if } tcp\_check(t) \\ \langle\rangle & \text{otherwise} \end{cases}$$

$$udp(a,pt,t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt)$$

$$http(a,pt,t) = restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt \wedge http\_check(pk.payload))$$

The following rules infer which security rules should be associated with addresses classified according to the predicates presented above:

$$\begin{aligned}
deploy_{block}(a,pt) &\leftarrow worm(a,pt) \\
deploy_{block}(a,pt) &\leftarrow botnet(a,pt) \\
deploy_{forward}(a) &\leftarrow \neg\exists pt : worm(a,pt) \vee botnet(a,pt) \\
deploy_{limit}(a,ip_limit) &\leftarrow dos(a) \\
deploy_{limit}(a,ip_limit) &\leftarrow port_scan(a) \\
deploy_{tcp}(a,pt) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge f.protocol = tcp \\
deploy_{udp}(a,pt) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge \\
&\qquad pt \neq 80 \wedge pt \neq 443 \wedge f.protocol = udp \\
deploy_{http}(a,80) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge f.dstport = 80 \\
deploy_{http}(a,443) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge f.dstport = 443 \\
deploy_{filter}(a,pm) &\leftarrow unsafe(a) \wedge\ danger(pm) \\
deploy_{inspect}(a) &\leftarrow unsafe(a)
\end{aligned}$$

Using the predicates *deploy* derived based on the flows, we now construct security

functions by composing elementary actions in parallel:

$$
\begin{aligned}
\mathit{stateless\_firewall}(t) = \quad & \bigcirc_{+}\{\, \mathit{forward}(a,t) : \mathit{deploy}_{\mathit{forward}}(a), a \in \textsc{Addr} \,\} \\
\circ_{+}\ & \bigcirc_{+}\{\, \mathit{block}(a,pt,t) : \mathit{deploy}_{\mathit{block}}(a,pt), a \in \textsc{Addr},\ pt \in \textsc{Port} \,\} \\
\mathit{ids}(t) = \quad & \bigcirc_{+}\{\, \mathit{limit}(a,n,t) : \mathit{deploy}_{\mathit{limit}}(a,n), a \in \textsc{Addr},\ n \in \mathbb{N} \,\} \\
\mathit{stateful\_firewall}(t) = \quad & \bigcirc_{+}\{\, \mathit{tcp}(a,pt,t) : \mathit{deploy}_{\mathit{tcp}}(a,pt), a \in \textsc{Addr},\ pt \in \textsc{Port} \,\} \\
\circ_{+}\ & \bigcirc_{+}\{\, \mathit{udp}(a,pt,t) : \mathit{deploy}_{\mathit{udp}}(a,pt), a \in \textsc{Addr},\ pt \in \textsc{Port} \,\} \\
\circ_{+}\ & \bigcirc_{+}\{\, \mathit{http}(a,pt,t) : \mathit{deploy}_{\mathit{http}}(a,pt), a \in \textsc{Addr},\ pt \in \textsc{Port} \,\} \\
\mathit{dpi}(t) = \quad & \bigcirc_{+}\{\, \mathit{inspect}(a,t) : \mathit{deploy}_{\mathit{inspect}}(a), a \in \textsc{Addr} \,\} \\
\mathit{dlp}(t) = \quad & \bigcirc_{+}\{\, \mathit{filter}(a,pm,t) : \mathit{deploy}_{\mathit{filter}}(a,pm), a \in \textsc{Addr},\ pm \in \mathcal{D} \,\}
\end{aligned}
$$

On the basis of these security functions we now define the chains to be deployed for filtering traffic generated by the target application by associating addresses to those chains corresponding to the classes to which the address belongs:

$$
\begin{aligned}
\mathit{safe\_chain} &= \mathit{stateless\_firewall} \circ_{\gg} \mathit{stateful\_firewall} \\
\mathit{unsafe\_chain} &= \mathit{stateless\_firewall} \circ_{\gg} \mathit{stateful\_firewall} \circ_{\gg} \mathit{dpi} \circ_{\gg} \mathit{dlp} \\
\mathit{dos\_chain} &= \mathit{stateless\_firewall} \circ_{\gg} \mathit{ids} \circ_{\gg} \mathit{stateful\_firewall} \\
\mathit{port\_scan\_chain} &= \mathit{dos\_chain} \\
\mathit{worm\_chain} &= \mathit{stateless\_firewall} \\
\mathit{botnet\_chain} &= \mathit{stateless\_firewall}
\end{aligned}
$$

Finally, we provide rewriting rules for converting security functions into Pyretic code. The argument t representing network traffic becomes implicit in Pyretic, which applies the transformation to concrete incoming traffic. The functions *DPIQuery*, *TCPFilter*, *UDPFilter*, and *HTTPFilter* exploit the rules of dynamic query that Pyretic provides. The overall security functions are obtained from the elementary

ones by using the combinators $\gg$ and $+$ of Pyretic that correspond to $\circ_{\gg}$ and $\circ_{+}$:

$$
\begin{aligned}
\mathit{forward}(a,t) &\rightsquigarrow \mathit{match}(\mathit{dstaddr}=a)\\
\mathit{block}(a,pt,t) &\rightsquigarrow \sim \mathit{match}(\mathit{dstaddr}=a,\ \mathit{dstport}=pt)\\
\mathit{limit}(a,n,t) &\rightsquigarrow \mathit{LimitFilters}(n,\mathit{dstaddr}=a)\\
\mathit{filter}(a,pm,t) &\rightsquigarrow \mathit{match}(\mathit{dstaddr}=a) \gg \mathit{RegexpQuery}(\mathit{regexp}(pm))\\
\mathit{inspect}(a,t) &\rightsquigarrow \mathit{match}(\mathit{dstaddr}=a) \gg \mathit{DPIQuery}\\
\mathit{tcp}(a,pt,t) &\rightsquigarrow \mathit{match}(\mathit{dstaddr}=a,\mathit{dstport}=pt) \gg \mathit{TCPFilter}\\
\mathit{udp}(a,pt,t) &\rightsquigarrow \mathit{match}(\mathit{dstaddr}=a,\mathit{dstport}=pt) \gg \mathit{UDPFilter}\\
\mathit{http}(a,pt,t) &\rightsquigarrow \mathit{match}(\mathit{dstaddr}=a,\mathit{dstport}=pt) \gg \mathit{HTTPFilter}
\end{aligned}
$$

To sum up, our approach consists in synthesising a program that encods the chain of security functions to be deployed in the network. To this end, we first learn the security properties to be guaranteed by the chain from the Markov automaton encoding the behavior of the application. These predicates are then used to derive the abstract specification of the chain to be deployed based on a constraint programming method. Finally this high level specification is used to generate the actual code of the concrete chain of security functions that can be either directly deployed in the network or be used for further optimizations.

1.7. Verifying correctness of chains

The next step consists in verifying correctness properties of security chains. As explained below, the chains generated by our method satisfy certain properties by construction.

Packet routing. Two desirable properties for packet routing are the absence of black holes and of loops. A black hole occurs when traffic is directed to a link where

no security function is installed. A loop is a cycle in the connections between security functions, such that network packets will be transmitted to a security function that they are already cleared.

Proposition 1.1: The synthesis of security chains described in section 1.6 avoids black holes and loops.

Proof. Our security functions and chains are constructed from elementary rules by parallel and sequential composition. In particular, each component of the chain is completely defined before being used, and there is no fixpoint construction or similar cyclic construct. This ensures that no black holes or cycles can exist at the high level of chain construction. We rely on the correctness of the translation to Pyretic to ensure that this property is preserved at the implementation level. □

Shadowing freedom and consistency. A security function is shadowing free if for any packet it contains at most one applicable rule.

Proposition 1.2: Security functions generated by the algorithm of section 1.6 guarantee shadowing freedom.

Proof. In the definition of *stateless_firewall*, shadowing would arise if for some address a and port pt, both rules $forward(a, t)$ and $block(a, pt, t)$ were composed in parallel. However, this is impossible because by definition the corresponding *deploy* predicates are mutually exclusive. Similarly, the different *deploy* predicates used in the definition of *stateful_firewall* are incompatible for any given address and port. □

We now show that our chains of security functions are consistent with the security properties determined on the basis of the traces t_{app} used for their generation.

Proposition 1.3: Given a trace t_{app} characterizing the network trafic generated by an application, the chain generated by the algorithm of section 1.6 forwards traffic classified as safe to the corresponding destinations but blocks or limits malicious traffic.

Proof. An address is considered as malicious if its t_{app} contains flows associated with the orgname of the address that are classified as worm, botnet, DoS, port scan or unsafe. Traffic directed to addresses considered as worm or botnet will immediately be blocked by the stateless firewall. Traffic towards addresses belonging to flows classified as DoS or port scan is transmitted to the IDS, which imposes a limit on the number of packets that will be allowed to pass.

Addresses associated with unsafe flows, i.e., network traffic that potentially compromise the confidentiality of private data, are handled by the DPI and DLP security functions that check for packet payload, according to the predicates *regexp* (associated with Android permissions) and *inspect_payload*. Encrypted traffic would have to be handled by specific inspection methods [42]. Traffic directed to IP addresses considered as safe is only subject to the stateless and stateful firewalls, which forward it and simply check conformance with the declared protocol. □

Beyond these structural correctness properties, we implemented techniques for verifying user-specified properties of both the control and the data planes of security chains [38]. These techniques build upon the Kinetic extension [21] of the Pyretic language that includes model checking capabilities for properties of the control plane, but they enable the verification of properties of the data plane as well.

The first technique is based on constraint solving. We encode elementary Pyretic actions as formulas in SMT-LIB, the input language of SMT solvers. For example, *identity* and *drop* are represented as *true* and *false*, and *match* and *modify* give rise to equational constraints on packet headers, where concrete IP addresses and port numbers are mapped to symbolic constants. Sequential and parallel composition correspond to conjunction and disjunction, and complement to negation. For example, Fig. 1.5 shows the encoding of a simple security chain as a logical formula. Data plane properties, such as whether certain packets are allowed to proceed or blocked, can then be verified by querying the constraint representing the chain.

The second technique is implemented based on symbolic model checking. In this case, Pyretic chains are represented as finite state machines. For this purpose, we

```
F1 = match(srcip=IP("198.122.37.15")) +
     match(srcip=IP("253.182.3.14"))
F2 = match(srcport=100) + match(srcport=200) + match(srcport=300)
F3 = match(srcport=400) + match(srcport=500) + match(srcport=600)
F4 = match(dstport=700) + match(dstport=800) + match(dstport=900)
chain = ((F1 >> F2) + (~F1 >> F3)) >> F4
```

$$
\begin{aligned}
\mathit{allowed} \equiv\ & \wedge \vee \wedge\ \mathit{srcip} = \mathit{ip0} \vee \mathit{srcip} = \mathit{ip1} \\
& \qquad \wedge\ \mathit{srcpt} = \mathit{pt1} \vee \mathit{srcpt} = \mathit{pt2} \vee \mathit{srcpt} = \mathit{pt3} \\
& \quad \vee \wedge \neg(\mathit{srcip} = \mathit{ip0} \vee \mathit{srcip} = \mathit{ip1}) \\
& \qquad \wedge\ \mathit{srcpt} = \mathit{pt4} \vee \mathit{srcpt} = \mathit{pt5} \vee \mathit{srcpt} = \mathit{pt6} \\
& \wedge\ \mathit{dstpt} = \mathit{pt7} \vee \mathit{dstpt} = \mathit{pt8} \vee \mathit{dstpt} = \mathit{pt9}
\end{aligned}
$$

Figure 1.5: A toy security chain in Pyretic and its encoding as a constraint.

extract strictly sequential subchains (such as $F1 \gg F2$ in the example of Fig. 1.5), and these give rise to state transitions that are guarded with conditions on header fields. A packet is accepted by the chain if there exists a path to the final state of the state machine all of whose transition conditions are satisfied, and this can be expressed using formulas of the CTL temporal logic and verified by the symbolic infinite-state model checker nuXmv [13]. This technique integrates well with the verification capabilities that exist in Kinetic, but extend them to encompass the data plane.

1.8. Optimizing security chains

When applying the techniques for generating security chains described in Section 1.6 for several applications, we obtain multiple chains that must be deployed in the network. However, many applications share certain services, such as for serving advertisements or for performing analytics, and the security chains corresponding to these applications are likely to contain similarities. Instead of simply combining chains using Pyretic's operator for parallel composition, or of deploying several chains using independent control planes (which could increase the overall vulnerability of the architecture), we aim at transforming several chains into a single one in a way that combines similar elements in different chains, minimizing the number of security functions and

rules. A security chain corresponds to a graph of security functions of different types such as firewalls, intrusion detection or data leakage prevention systems [26]. In turn, a security function consists of a set of security rules applied in parallel, where a rule is described by a guard and an action.

Our transformation, presented in [40], is based on two procedures. The procedure *merge_functions* takes two security functions (assumed to be of the same type) as inputs and merges them. Rules of either of the two functions whose guards are disjoint from the rules of the other function cannot be conflicting and are simply added to the merged function. For guards that appear in both security functions, if the associated action is the same, the rule is again added to the result. In case of different actions, we rely on priorities provided by the network operator in order to determine which rule to include in the result. The procedure *merge_chains* composes two chains. It first identifies security functions of the same type that appear in the input chains and merges them using *merge_functions*, while functions that have no equivalent in the other chain are simply added to the resulting chain. The edges of the output chain mirror those of the input chains. A quantitative evaluation is provided in Section 1.9.

The transformations *merge_functions* and *merge_chains* do not involve structural modifications of the chains and therefore preserve the structural properties stated in propositions 1.1 and 1.2. The consistency with classification (proposition 1.3) may not be preserved when an address is classified differently by the flows collected for two different applications. In our experiments based on existing benchmarks, we have never observed this happening.

The second aspect of optimization concerns the deployment of security chains in the SDN network. The placement of security functions in a network has to satisfy certain constraints: the order in which functions appear in the chain has to be respected, the number of rules deployed on any given switch must not exceed the capacity of that switch, and the capacity of channels connecting switches must be respected. Within these constraints, we aim at optimizing metrics such as the number of required switches, the congestion of the service, and its probability of availability.

In order to make the optimization problem feasible using standard solvers, we aggregate destination addresses, as well as network resources in our model. In our context, we can consider collections of IP addresses that will be associated with SDN switches and then assign the rules for these collections of destinations to the corresponding network equipment. We call these collections of IP addresses *destination aggregates*. It is important for the optimality of the placement that destination aggregates represent comparable traffic load. Thus, we compute the destination aggregates as the result of a knapsack problem. The number of knapsacks is computed as the ratio between the overall traffic load and the capacity of the smallest channel in order to guarantee that we will be able to place every destination aggregate on every channel.

We also aggregate switches as network paths, i.e. as sequences of switches connected in line without branching. The properties of network paths are computed depending on the properties of their internal switches and channels. We will consider the following properties in the remainder of this chapter:

- *length*: the number of switches connected in sequence,
- *rule_capacity*: the minimal rule capacity in the path,
- *load_capacity*: the minimal load capacity in the path, and
- *path_probability*: the availability probability of the path.

The information describing the chains and the network are provided as input for the placement. Destination aggregates are represented by the set *dests*. The number of flows to handle per destination aggregate is represented by a dictionary *dest_load* indexed by the set *dests*. In a similar manner, we introduce a dictionary *dest_weight* that associates with each destination the number of aggregated IP addresses. The dictionary *function_weight* associates with each security function its number of rules per destination. For each security function, our synthesis algorithm guarantees that a destination will be protected by exactly two rules, one for incoming trafic and one for outgoing trafic. Network paths are represented by the set *paths*. The dictionary *path_length* provides information about the length of each path, *rule_capacity* asso-

ciates with each path the smallest rule capacity among the switches on that path, *load_capacity* stores the load capacity of each path, and *path_probability* indicates the availability probability of each path. The relation *path_connection* indicates if a path is the successor of another path. Finally, we derive two sets *incomings* and *outgoings* which represent incoming and outgoing paths of the network. Namely, $i \in incomings$ if and only if $\forall p \in paths,\ path\_connection_{(p,i)} = 0$ and $o \in outgoings$ if and only if $\forall p \in paths,\ path\_connection_{(o,p)} = 0$.

We represent the placement of rules by the variables *dest_placement*, a matrix of binary variables indexed by *paths* and *dests* that indicate whether the rules concerning a destination d are placed on a path p and the array *used_path* of binary variables that identify used network paths. The following constraints must be respected for a placement to be valid:

1. Constraints on path usage:
   (a) A path is used if the rules for at least one aggregate of destinations are placed on it.
   $\forall p \in paths,\ |dests| \times used\_path_p \geq \sum_{d \in dests} dest\_placement_{(p,d)}$
   (b) A path can be used only if at least one of its successors is used.
   $\forall p \in paths,\ |dests| \times used\_path_p \geq \sum_{suc \in paths} path\_connection_{(p,suc)} \times used\_path_{suc}$
   (c) The symmetric constraint requiring that a path can be used only if at least one of its predecessors is used.
2. Constraints on destination placement:
   (a) Each destination must be placed on at least one incoming path.
   $\forall d \in dests,\ \sum_{p \in incomings} dest\_placement_{(p,d)} \geq 1$
   (b) The symmetric constraint requiring that each destination must be placed on at least one outgoing path.
3. Capacity constraints:

(a) Constraints on the rule capacity of each path in the network.

$$\forall p \in paths,\ rule\_capacity_p \ \geq$$
$$function\_weight \times \sum_{d \in dests} dest\_weight_d \times dest\_placement_{(p,d)}$$

(b) Constraints in terms of traffic load of each path in the network.

$$\forall p \in paths,\ load\_capacity_p \geq \sum_{d \in dests} dest\_load_d \times dest\_placement_{(p,d)}$$

We want to optimize several objectives while ensuring the above constraints: (i) network utilization, i.e., the number of switches needed for deploying the security chains, (ii) service congestion due to the concentration of traffic load on a few channels and (iii) probability of availability, i.e., the probability for the service to be available and not affected by network downtimes. In our case, these three criteria are combined in a single objective function to minimize. Results of experiments with non-linear solvers, linear approximations, and optimizing SMT solvers are described in Section 1.9.

1.9. Performance evaluation

We implemented the techniques described in this chapter in a prototype consisting of 13457 lines of Python 2.7 and 111 lines of SWI-Prolog (v7.6.4) and evaluated them on a Macbook Pro (13-inch, 2017) with an Intel® core i7 processor (2.5 GHz) and 16 GB RAM. The back-end solvers used for verification (Section 1.7) are the model checker `nuXmv` (v1.0.1) and the SMT solvers `cvc4` (v1.5) and `veriT` (v201506). For optimization (Section 1.8) we employed the simplex solver `glpsol` (v4.64), the MINLP solver `couenne` (v0.5.6) and optimization module of the SMT solver `z3` (v4.8.0). During our experiments we considered 10 Android applications given in Table 1.1. For each application we indicate the number of recorded flows, the corresponding number of IP addresses, the presence of a manifest file and the number of requested permissions.

In our experiments we evaluated the following criteria: the complexity of the chains (numbers of security functions and rules), the response times for synthesis, factoriza-

Table 1.1: The set of Android applications considered for evaluation.

Applications	Flows	Addresses	Manifest	Permissions	Functions	Rules
disneyland	282	5	no	–	4	44
dropbox	1000	17	yes	5	5	311
faceswitch	151	30	yes	3	5	425
lequipe	1000	151	no	–	4	1640
meteo	1000	80	no	–	4	716
ninegag	1000	88	no	–	4	930
pokemongo	275	24	yes	6	5	485
ratp	779	3	no	–	4	28
skype	1000	161	yes	11	5	6529
viber	1000	78	yes	15	5	4163

tion and verification, the accuracy with which security chains detect attacks, and the overhead incurred by deploying chains in a network.

Complexity of security chains. Table 1.1 shows the numbers of security functions and rules of the chains generated for the different applications. Each chain contains either 4 or 5 security functions, depending on the presence of the manifest file, which causes DLP rules to be generated. The number of rules clearly illustrates the high disparity of network behavior observed for the applications.

Table 1.2: Number of rules for combined chains.

Nb. of apps	Parallel Composition	Combined Generation	Chain Merging
1	311	311	311
2	1951	3987	1947
3	2376	6033	2367
4	2420	6153	2407
5	3136	8289	3119
6	3164	8361	3143
7	9693	25949	9667
8	13856	51041	13825
9	14341	61181	14305
10	15271	71147	15231

Table 1.2 shows the number of rules corresponding to a chain obtained by successively combining chains for individual applications. We compare three different

approaches: parallel composition simply composes individual chains using Pyretic's + operator, combined generation generates a single chain from the concatenation of the flows corresponding to the applications, and chain merging implements the algorithm presented in Section 1.8. The results show that parallel composition and merging produce significantly fewer rules than a combined generation. In contrast to parallel composition where the number of overall functions corresponds to the sum of the numbers of functions per application, merging preserves the number of security functions to be deployed, reducing overhead and attack surface.

Response times. In our experiments, the time needed for learning the behavior of an application from a recorded trace is on the order of minutes, whereas generating and merging chains takes at most a few seconds. For example, merging the security chains for the ten applications in our benchmark set takes 5 seconds. These numbers clearly illustrate the fact that learning the Markov automaton representing an application is not feasible at runtime. However, assuming that applications are relatively stable, learning can be done offline, and the cost of a learning session can be amortized over time. In our overall architecture, we suggest that security chains corresponding to applications be stored in a database. Given the applications to protect, we can at deployment time load the corresponding chains, merge them, and install the result through the SDN controller.

In order to evaluate the performance of formally verifying properties of chains, we artificially generated chains whose numbers of rules varied between 1,000 and 10,000 [38]. The SMT-based verification technique results in linear growth with about 15 seconds for the largest chains, whereas `nuXmv` exhibits super-linear growth and requires more than 40 seconds for the largest chains. However, `nuXmv` performed better for long chains with many security functions composed sequentially. In both cases, these numbers indicate that formal verification is feasible as an off-line task.

Accuracy of security chains In order to evaluate the accuracy of the generated chains, we used 70% of the recorded flows for an application for generating the chain

and then used the remaining 30%, into which we injected a simple port scan, for evaluating its accuracy. We measured accuracy as the ratio between the sum of true positives and true negatives by the total number of flows. We also fixed a threshold, varying between 0 and 10, corresponding to the number of attack flows that must be analyzed before blocking the traffic. Table 1.3 shows the minimal, maximal and average accuracy observed for each chain of security functions. We also computed the corresponding results for the combined chain for all 10 applications in order to observe a potential loss of accuracy, but obtained identical values.

Table 1.3: Accuracy of chains generated for protecting applications.

Applications	Avg. Accuracy	Min. Accuracy	Max. Accuracy
viber	0.683	0.502	0.997
faceswitch	0.812	0.518	0.990
dropbox	0.997	0.993	1.000
ninegag	0.509	0.498	0.526
disneyland	0.992	0.986	1.000
pokemongo	0.743	0.512	0.994
skype	0.998	0.998	0.998
lequipe	0.518	0.496	0.537
meteo	0.837	0.510	0.998
ratp	0.940	0.692	0.999

The results are mixed, depending on the considered application. For certain applications, the 30% of logged flows used for the evaluation only contain flows that were already encountered during the learning phase, and we obtain an accuracy close to 100% while for other applications the recorded flows have stronger disparity. These results indicate that the quality of the data used for learning is important. We believe that our approach is acceptably stable, since the orgnames of servers contacted by an application should not change in between major updates. We also believe that the definitions of the predicates that we use for classifying attacks are probably quite naive. Since our approach makes it easy to plug different definitions into our algorithm for chain synthesis, one can experiment different rules without modifying the overall architecture.

Overhead incurred by deploying security chains. In order to evaluate the cost in terms of bandwidth related to deploying our security chains, we simulated the traffic generated by each application with and without the corresponding chains and measured the resulting bit rate. The results of these experiments are presented in Fig. 1.6.

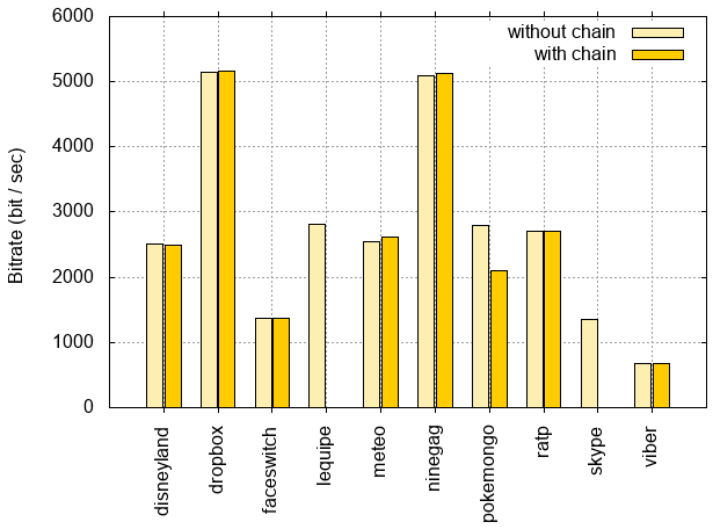

Figure 1.6: Overhead in terms of bandwidth introduced by security chain deployment.

In contrast to the other evaluations described here, we used a Frenetic implementation of security chains because the Pyretic language is no longer supported by modern SDN controllers. For most applications, the overhead is negligible. The observed differences are minor, probably due to the underlying OVS switches and their dictionary-based flow tables. However, we were unable to deploy the chains for two applications (lequipe and skype) because the Frenetic controller generated too many rules by compiling our chains into OpenFlow. Because our approach is agnostic to implementation languages, it could easily be extended by new implementations based on P4 or involving Network Function Virtualization.

1.10. Conclusions

This chapter introduces a method for automating the orchestration of security functions driven by process learning, and illustrates how it could be used for protecting Android devices by relying on software-defined networks. It contributes to bridging the gap between learning and verification techniques.

The method that we propose addresses four main problems: (i) modeling the specific security needs of applications through process learning techniques, (ii) generating corresponding chains of security functions based on methods of formal synthesis, (iii) verifying the correctness properties of these chains, and (iv) optimizing their deployment by merging chains and adapting them to the network infrastructure. We evaluated the performance of the method through extensive series of experiments.

The flexibility of SDN infrastructures enables synthesizing and deploying security chains that are specific to the networking behavior of individual applications running on smart devices. By construction, the obtained chains ensure certain correctness properties, and specific properties can be formally verified based on SMT solving and model checking. Finally, by applying appropriate optimization methods, the impact of deploying security chains on network performance can be substantially reduced.

This work opens several directions for future research. A closer coupling of network and system aspects could be investigated, beyond the generation of regular expressions based on the permissions declared in manifest files of applications. Emerging methods from explainable artificial intelligence could also be considered for facilitating the interpretation of automation results, together with the use of more elaborated detection techniques. Finally, it could interesting to explore complementary synthesis techniques for taking into account the dynamics of attacks, for instance with more sophisticated models expressed in temporal logic, by following a similar overall methodology.

Bibliography

[1] Android Permissions System. https://developer.android.com/guide/topics/security/permissions.html.

[2] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications (INFOCOM'04)*, 2004.

[3] Zafar Ayyub and Rui Miao. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM Computer Communication Review*, 2013.

[4] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. On Demystifying the Android Application Framework: Re-visiting Android Permission Specification Analysis. In *Proceedings of the 25th USENIX Security Symposium (NSDI 2016)*, 2016.

[5] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards Verifying Controller Programs in Software-Defined Networks. In *Proc. 35th ACM SIGPLAN Intl. Conf. Programming Language Design (PLDI'14)*, pages 282–293, Edinburgh, UK, 2014.

[6] Dominique Barthel, JP Vasseur, Kris Pister, Mijeom Kim, and Nicolas Dejean. Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks. RFC 6551, March 2012.

[7] Ryan Beckett. *Network Control Plane Synthesis and Verification.* PhD thesis, University of Princeton, 2018.

[8] Carlos J. Bernardos, Akbar Rahman, Juan-Carlos Zúñiga, Luis M. Contreras, Pedro Andres Aranda, and Pierre Lynch. Network Virtualization Research Challenges. RFC 8568, April 2019. URL https://rfc-editor.org/rfc/rfc8568.txt.

[9] Ivan Beschastnikh, Jenny Abrahamson, Yuriy Brun, and Michael D. Ernst. Synoptic: Studying Logged Behavior with Inferred Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 448–451, New York, NY, USA, 2011. ACM.

[10] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms. In *IEEE Transactions on Software Engineering*, volume 41, pages 408–428, 2015.

[11] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsch. *Handbook of satisfiability*. IO press, 2008.

[12] A.W. Biermann and J.A. Feldman. On the Synthesis of Finite-State Machines from Samples of Their Behavior. In *IEEE Transactions on Computers*, 1972.

[13] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Proc. 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 334–342, Vienna, Austria, 2014. doi: 10.1007/978-3-319-08867-9_22.

[14] Benoit Claise. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101, January 2008.

[15] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2016.

[16] Joseph Dilip and Stoica Ion. Modeling Middle Boxes. In *IEEE Network: The Magazine of Global Internetworking archive*, 2008.

[17] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Ra-

jarajan. Android Security, a Survey of Issues, Malware Penetrations and Defenses. In *IEEE Communications Surveys & Tutorials*, 2015.

[18] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN, an Intellectual History of Programmable Networks. *SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[19] Nate Foster, Michael J. Freedman, Rob Harrison, Christopher Monsanto, and David Walker. Frenetic, a Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011)*, 2011.

[20] Nate Foster, Jedidiah McClurg, Hossein Hojjat, and Pavol Cerny. Efficient Synthesis of Network Updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, 2015.

[21] Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Kata, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Rexford Jennifer, Cole Schlesinger, Alec Story, and David Walker. Languages for Software-Defined Networks. In *Software Technology Group*, 2016.

[22] GData. Mobile Malware Report. http://www.gdatasoftware.com, 2019.

[23] Mark J. Handley, Eric Rescorla, and Internet Architecture Board. Internet Denial-of-Service Considerations. RFC 4732, December 2006.

[24] Hongxin Hu, Wonkyu Han, Gail Joon Ahn, and Ziming Zhao. Flowgard : Building Robust Firewalls for Software-Defined Networks. In *Proceedings of the third workshop on Hot topics in software defined networking (SIGCOMM 2014)*, 2014.

[25] Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, and Olivier Festor. Towards Cloud Based Compositions of Security Functions for Mobile Devices. In *IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*, 2015.

[26] Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, and Olivier Festor. Behavioral and Dynamic Security Functions Chaining for Android Devices. In *Proceedings of the 11th IFIP/IEEE/ACM SIGCOMM International Conference on Network and Service Management (CNSM 2015)*, 2015.

[27] Mi-Young Kang, Jin-Young Choi, Inhye Kang, Hee Hwan Kwak, So Jin Ahn, and Myung-Ki Shin. *A Verification Method of SDN Firewall Applications.* IEICE Transactions on Communications, 2016.

[28] Ahmed Khurshid, Xuan Zou, Winxuan Zhou, Matthew Caesar, and P. Brighten. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the first Workshop on Hot Topics in Software-Defined Networks (HotSDN'12)*, 2012.

[29] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhamad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI 2015)*, 2015.

[30] Jeongmin Kim, Hyunwoo Choi, Hun Namkung, Woohyun Choi, Byungkwon Choi, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. Enabling Automatic Protocol Behavior Analysis for Android Applications. In *Proceedings of the 12th International Conference on Emerging Networking EXperiments and Technologies (CONEXT 2016)*, pages 281–295, New York, NY, USA, 2016.

[31] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A Survey on Security for Mobile Devices. In *IEEE Communications Surveys & Tutorials*, 2012.

[32] Abdelkader Lahmadi, Frederic Beck, Eric Finickel, and Olivier Festor. A Platform for the Analysis and Visualization of Network Flow Data of Android Environments. IFIP/IEEE Intl. Symposium on Integrated Network Mgmt (IM 2015).

[33] Gary S. Malkin and Tracy LaQuey Parker. Internet Users' Glossary. RFC 1392, January 1993.

[34] Andrew Newton, Byron Ellacott, and Ning Kong. HTTP Usage in the Registration Data Access Protocol (RDAP). RFC 7480, March 2015. URL https://rfc-editor.org/rfc/rfc7480.txt.

[35] Andrés F. Ocampo, Juliver Gil-Herrera, Pedro H. Isolani, Miguel C. Neves, Juan F. Botero, Steven Latré, Lisandro Zambenedetti, Marinho P. Barcellos, and Luciano P. Gaspary. Optimal Service Function Chain Composition in Network Functions Virtualization. In *Proceedings of the IFIP International Conference on Autonomous Infrastructure, Management and Security (IFIP AIMS'17)*, pages 62–76. Springer International Publishing, 2017.

[36] Nikolaos Petroulakis, Konstantinos Fysarakis, Ioannis Askoxylakis, and George Spanoudakis. Reactive Security for SDN/NFV-enabled Industrial Networks leveraging Service Function Chaining. *Transactions on Emerging Telecommunications Technologies*, 12 2017. doi: 10.1002/ett.3269.

[37] Nicolas Schnepf. *Orchestration et Vérification de Fonctions de Sécurité pour des Environnements Intelligents.* PhD thesis, University of Lorraine, 2019.

[38] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated Verification of Security Chains in Software-Defined Networks with Synaptic. In *Proceedings of the 3rd IEEE Conference on Network Softwarization (IEEE NetSoft 2017)*, 2017.

[39] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Towards Generation of SDN Policies for Protecting Android Environments based on Automata Learning. In *Proceedings of the 16th Network Operations and Management Symposium (IEEE/IFIP NOMS 2018)*, 2018.

[40] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated Factorization of Security Chains in Software-Defined Networks. In *Proceedings of the 16th IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*, 2019.

[41] Alireza Shameli-Sendi, Yosr Jarraya, Makan Pourzandi, and Mohamed Cheriet. Efficient Provisioning of Security Service Function Chaining Using Network Security Defense Patterns. *IEEE Transactions on Services Computing*, PP, 10 2016. doi: 10.1109/TSC.2016.2616867.

[42] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the ACM Conference on Special Interest Group on Data Communication (SIGCOMM 2015)*, pages 213–226, New York, NY, USA, 2015. ACM.

[43] Anna Sperotto. *Flow-based Intrusion Detection.* PhD thesis, University of Twente, 2010.

[44] Xuetao Wei. ProfileDroid : Multi-layer Profiling of Android Applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (MOBICOM 2012)*, 2012.