Rule-Based Synthesis of Chains of Security Functions for Software-Defined Networks

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, Stephan Merz

Rule-Based Synthesis of Chains of Security Functions for Software-Defined Networks

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, Stephan Merz

Université de Lorraine, CNRS, Inria, Loria, F-54000 Nancy, France
nicolas.schnepf@inria.fr

Abstract. Software-defined networks (SDN) offer a high degree of programmability for handling and forwarding packets. In particular, they allow network administrators to combine different security functions, such as firewalls, intrusion detection systems, and external services, into security chains designed to prevent or mitigate attacks against end user applications. These chains can benefit from formal techniques for their automated construction and verification. We propose in this paper a rule-based system for automating the composition and configuration of such chains for Android applications. Given the network characterization of an application and the set of permissions it requires, our rules construct an abstract representation of a custom security chain. This representation is then translated into a concrete implementation of the chain in Pyretic, a domain-specific language for programming SDN controllers. We prove that the chains produced by our rules satisfy a number of correctness properties such as the absence of black holes or loops, and shadowing freedom, and that they are coherent with the underlying security policy.

Keywords: Security Management, Software-Defined Networking, Android, Rule-Based Programming.

1 Introduction

Software Defined Networking (SDN) is a recent paradigm in the field of network management and security that promises to improve network programability by decoupling the data and control planes. In this context, the data plane consists in virtual switches and equipment responsible for forwarding traffic across the network, whereas the control plane consists in a single or multiple controllers, responsible for dynamically adjusting the configuration of the data plane in response to network events. The communication between the two planes is supported by a dedicated protocol, typically OpenFlow. SDN is oftenly used in conjunction with Network Function Virtualization (NFV) for supporting the automated deployment of more advanced security functions running in virtual machines in addition to virtual switches.
Based on these new networking paradigms, researchers proposed the concept of chains of security functions for protecting end user applications

against attacks. These chains are composed of different security mechanisms such as intrusion detection systems (IDS) or firewalls in cloud infrastructures for the protection of end users. The programmability introduced by software defined networks helps greatly for automating the configuration, adjustment, and deployment of chains of security functions. Programming SDN controllers is made simpler by the introduction of domain-specific programming languages such as Pyretic designed for implementing chains of security functions at a high level of abstraction before compiling them into low-level OpenFlow configuration rules. Nevertheless, the validation of chains of security functions remains non-trivial: the complexity of their internal security functions makes it incredibly easy to introduce misconfigurations and security holes that can then be exploited by attackers targeting these networks and their users. There exist several approaches for formally verifying chains of security functions a posteriori (cf. section 2). In this paper we propose an inference system based on Horn clauses for synthesizing chains of security functions in an automated manner, ensuring that the generated chains satisfy elementary correctness requirements. We specifically target the protection of network traffic generated by Android applications, taking into account security requirements derived from the observed networking behavior of an application and the set of operating-system level requirements that the application requires. We rely on our previous work [22] for characterizing the networking behavior by learning a finite Markov chain that represents the network flows observed by a dedicated network probe. Using an inference system in order to construct a high-level representation of the security chain, we obtain a declarative description of the generation process that simplifies reasoning about the properties it guarantees, such as the consistency of the deployed security rules and the absence of loops.

Our contributions are threefold: (i) we design a system of Horn clauses for the inference of chains of security functions that ensure certain correctness properties and that can be translated directly into Pyretic implementations, (ii) we propose a new representation of security requirements of Android end users, (iii) we have implemented a prototype of our method in Prolog.

The remainder of this paper is organized as follows. Section 2 gives an overview of existing related work. Section 3 introduces background on network security. Section 4 describes the system of Horn clauses used for synthesizing SDN based chains of security functions. Section 6 discusses the correctness properties that are guaranteed by our solution. Section 7 concludes and points out future research perspectives.

2 Related work

Much work has been directed towards the detection of attacks: most approaches rely on packet analysis for detecting attacks [1,9,6]: these methods provide good accuracy; nevertheless the increasing mass of traffic to analyze in modern network require more efficient detection methods. In her PhD thesis, Anna Sperotto proposed to base the detection of attack

on flow records instead of packet traces [23]: the objective of her work is to configure IDS depending on observed traces of attacks. Nevertheless, she does not consider data transmitted during exchanges and does not explore the possibility of deploying other security functions than IDS. Comparing detection methods requires dedicated datasets such as the CTU 13 [13] which contains botnet traces, but further datasets are necessary in order to capture other types of attacks such as DOS, port scanning or worm attacks.

New perspectives for the protection of end users are introduced by the SDN paradigm: one possible approach consists in composing different security functions into chains [16,15]. There exists various work in the literature that addresses the formal verification of such chains. For instance, Vericon [3] is a framework in which properties to be guaranteed by network policies can be expressed and verified. Al-Shaer and Hamed [2] propose another approach for the discovery of anomalies in distributed firewalls, targeting in particular contradictions in large firewall policies. Those approaches focus on the verification of the data plane of chains of security functions but do not cover the validation of aspects related to the control plane. In addition, while these approaches are useful for validating a posteriori the correctness of policies w.r.t. pre-established criteria, they do not generate a policy according to these criteria, and they may miss configuration errors that are not covered by the specified properties.

We previously [21] introduced the Synaptic checker for the verification of both control and data plane properties of an SDN policy. This checker relies on the Pyretic programming language [11], part of the Frenetic family of languages for programming SDN controllers [12]. Pyretic is implemented as a domain-specific language embedded in Python for describing chains of rules at a level of abstraction well above that of actual SDN protocols, but from which OpenFlow rules can be compiled. Pyretic is complemented by an extension, called Kinetic, for describing control plane policies; Kinetic also offers formal verification techniques based on model checking [17]. Synaptic extends this formalism for verifying the correctness of both the control and data planes of Pyretic policies, before their deployment in the network. We also proposed [22] an extension of the Synaptic checker with features for automatically learning network behaviors, represented as a Markov chain, of Android applications using their network flow traces. Our present paper is based on this technique and exploits it for the automatic generation of SDN policies satisfying the security requirements corresponding to such an application.

Most methods for the validation of chains of security functions consider their correctness a posteriori, using techniques such as model checking [8,10] or SMT solving [5]. In this paper, we suggest a declarative technique for the automatic synthesis of such chains, expressed at a high level of abstraction. We express our technique in terms of Horn clauses and have prototypically implemented it in Prolog. This representation makes it easy to modify the rules in order to take into account varying end-user requirements, rather than hard-coded policies defined by operators. It also helps for formally establishing a priori certain properties that the generated chains ensure.

3 Background on network security

We introduce some background regarding network security, with respect to the attacks we consider, network programmability, Android environments, and profiling of applications.

3.1 Network flows and considered attacks

Our work is centered on the inference of chains of security functions based on a characterization of end-user applications in terms of network flows. According to RFC 5101 [7], network flows can be defined as "collections of IP packets passing through an observation point in the network during a certain interval". They are generally described by different properties such as IP version, source and destination IP addresses (*srcaddr* and *dstaddr*), source and destination ports (*srcport* and *dstport*), network protocol (*protocol*), and the numbers of packets and bytes (*packets* and *bytes*). In our context, they are collected directly on end-user devices [18], and are extended with a timestamp (*timestamp*) and the name of the source application (*appname*). We furthermore complement this information with the name of the organization (*orgname*) responsible for the network that contains the destination IP address. As highlighted in [23], network flows are widely used for the detection of different categories of attacks, especially denial of service, port scanning, worms and botnets.

A *denial of service* (DoS) attack is characterized by "one or more machines targeting a victim and attempting to prevent the victim from doing useful work" [14]. In our context, we will consider DoS attacks that are observable from a networking point of view, such as SYN flood attacks where a high number of SYN packets are sent to the same host in order to overload the TCP stack with open connections that will never been closed. In a *port scanning* attack, an application initiates connections with a wide range of ports of a machine (or several machines) in order to detect which ports are open. We will consider port scanning techniques such as those generated by the `nmap` port scanner, available on standard Linux platforms. A *worm* is a program that can run independently, will consume the resources of its host in order to maintain itself, and can propagate a complete working version of itself to other machines [19]. Worms replicate by exploiting the vulnerabilities of applications and operating systems or by social engineering methods. We will consider worms that scan a certain port on many different machines in order to exploit a specific vulnerability on operating systems. A (potentially) *malicious bot* is a program that is installed on a system in order to enable that system to automatically (or semi-automatically) perform a task (or a set of tasks) typically under the command and control of a remote administrator, called "bot master" [4]. These bots can be detected based on the high volume of traffic they exchange with their controller or possibly by the use of network protocols that are not commonly observed in a given context.

3.2 Network programming with Pyretic

Pyretic is a domain-specific programming language embedded in Python for the configuration of SDN controllers. It is based on fundamental blocks (called *policies*) that can be combined for generating more complex policies. In the remainder of this paper we will consider the following primitive policies and composition operators:

- *identity*: forward all packets;
- *drop*: block all packets;
- *match*$(x_1 = y_1, \ldots, x_n = y_n)$: forward those packets whose header fields x_i contain the values y_i;
- *modify*$(x_1 = y_1, \ldots, x_n = y_n)$: forward all packets but modifies their header fields x_i to the values y_i;
- *count_packets*$(x_1 = y_1, \ldots, x_n = y_n)$: count the number of packets whose header fields x_i contain the values y_i;
- *LimitFilters*$(k, x_1 = y_1, \ldots, x_n = y_n)$: forward a maximum of k packets whose header fields x_i contain the values y_i;
- *RegexpQuery*(*pattern*): forward packets whose payload matches the given pattern (a regular expression);
- *sequential*$(p_1, \ldots, p_n)$: compose the policies $\{p_1, \ldots, p_n\}$ in sequence, also written $p_1 \gg p_2$ for $n = 2$;
- *parallel*$(p_1, \ldots, p_n)$: compose the policies $\{p_1, \ldots, p_n\}$ in parallel, also written $p_1 + p_2$ for $n = 2$;
- *negate*(p): forward packets that are blocked by the policy p, and block those that are forwarded by p, also written $\sim p$.

This language and its primitives will serve as a support for building and composing security functions for software-defined networks.

3.3 Focus on Android environments

We will target the protection of Android devices and their applications. In particular, we will rely on the Flowoid probe [18] dedicated to Android devices and will use it for exporting network flow records of applications running on these devices. Given its position as the market-leading operating system for smart devices and the limited effectiveness of preventive methods for proactively detecting malware applications, Android is particularly exposed to security attacks. For instance in 2016, Kaspersky Lab identified more than 3.5 million malware apps on the Google market store. In addition to the network flows previously mentioned, we will take into account the permissions that an application holds for accessing resources. An Android application must explicitly state the permissions it requires in its manifest file, and the security system of Android distinguishes between normal and (potentially) dangerous permissions. The former represents accesses to resources considered as non-critical, and they are automatically granted when requested. Dangerous permissions provide access to critical information such as user contacts, and they must be granted manually, when the application is installed. An application holding such a permission could use it to leak sensitive data to remote malicious servers. Therefore, the security chains that we generate include specific checks to prevent such attacks from occurring.

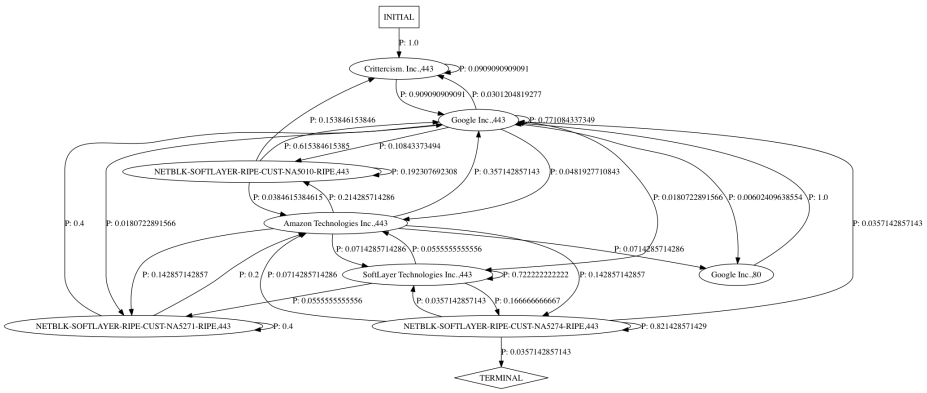

Fig. 1. Automaton describing the behavior of an Android application.

3.4 Automated network profiling of Android applications

We proposed in [22] an algorithm for automatically learning a Markov model of the networking behavior of Android applications: this model is based on trace logs of the flows generated by an application, collected directly on the device by using the Flowoid probe [18] that can associate flows with applications depending on the name of the package producer. These flow records are then transmitted to a centralized platform by using the NetFlow protocol in order to learn the communication pattern of the application: the first stage of our learning algorithm consists in aggregating destination IP addresses of flows depending on common owning organizations. This information is obtained from the result of *whois* requests, more specifically from the *orgname* and *netname* fields of the answers to such requests. Once flows have been completed by this information we build our Markov model of the application in the following manner: states are computed as collections of flows sharing the same orgname or netname; transitions are computed depending on the successions of orgnames or netnames in the input log; finally, probabilities are computed depending on the number of flow records in the input state and on the number of occurrences of the transition in the input log. Algorithm 1 shows the pseudo-code for the learning algorithm, and Fig. 1 contains the automaton obtained for the application Pokemon Go from a flow trace containing 150 flow records. In summary, the automaton represents the connections that the application establishes to addresses associated with different organizations.

4 Automated synthesis of chains of security functions

We propose in this paper a strategy based on rule-based programming for the automated inference of chains of security functions, based on a char-

Algorithm 1 Automaton learning algorithm.

```
States := ∅
Transitions := ∅
Flows :=List of flows.
                                          ▷ Initialize the set of states
flow := Flows[0]
States[flow] := 1
                                  ▷ Count occurrences of states and transitions
for i ∈ 1..N do
    transition := (flow, Flows[i])
    flow := Flows[i]
    if flow ∈ States then
        States[flow] += 1
    else
        States[flow] := 1
    end if
    if transition ∈ Transitions then
        Transitions[transition] += 1
    else
        Transitions[transition] := 1
    end if
end for
                                  ▷ Compute the probabilities of transitions
for transition ∈ Transitions do
    Transitions[transition] := Transitions[transition]/States[transition.srcState]
end for
```

acterization of an Android application in terms of its network behavior and the permissions it holds. Our approach is based on the classification of the network traffic generated by an application as described in Sect. 3.4. We then use logic programming for deriving the functional specification of the security chain to be deployed, and finally generate an instance of such a chain using the Pyretic language for programming SDN controllers. Our work is presented in the context of Android protection, although it can be extended to any systems using similar permissions for protecting user data. Similarly, it is possible to consider other formats or implementations for the generation of chains of security functions, for instance by exploiting Network Function Virtualization (NFV) facilities.

4.1 Representing flows, traces, and security functions

Recall that flows are collections of packets sharing certain properties. We summarize a flow in a record that contains the key attributes of a flow. In the following, $\mathbb{N}$ and $\mathbb{R}^+$ denote the sets of natural and non-negative real numbers, $\textsc{Addr} = \{0,1\}^{32}$ and $\textsc{Port} = \{1,\ldots,65535\}$ the sets of IP addresses and ports, $\textsc{Prot} = \{TCP, UDP, ARP, ICMP, \ldots\}$ the set of networking protocols and $\textsc{String}$ the set of (ASCII) character strings.

Definition 1. A *flow* f is a record that contains the following attributes and maps them to values in the corresponding domains:

$$\begin{aligned} f.timestamp &\in \mathbb{R}^+ & f.srcaddr &\in \textsc{Addr} \\ f.dstaddr &\in \textsc{Addr} & f.srcport &\in \textsc{Port} \\ f.dstport &\in \textsc{Port} & f.bytes &\in \mathbb{N} \\ f.packets &\in \mathbb{N} & f.protocol &\in \textsc{Prot} \\ f.appname &\in \textsc{String} & f.orgname &\in \textsc{String} \end{aligned}$$

A *flow trace* is a sequence of flows such that time stamps are strictly increasing along the sequence. By abuse of notation, we will write $f \in t$ to indicate that flow f appears as an element of the sequence t. 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. For an application *app* we will note t_{app} a flow trace such that $f.appname = app$ for all flows f in the trace, and P_{app} the list of permissions it requests. We let $\mathcal{D}$ stand for the set of Android permissions qualified as dangerous. □

Our approach aims at constructing a chain of security functions for protecting a specific application. Security functions transform network traffic, i.e. sequences of packets. Packets contain header fields similar to flows, except those that contain aggregate information such as *packets* and *bytes*, but they also contain a packet payload (field *payload*) that represents the actual information transmitted in a packet. We overload

the merge operation $\oplus$ to apply to sequences of packets as well as to flow traces.
Security functions are built by combining in parallel basic building blocks, called *security rules*, and they in turn give rise to chains by applying parallel or sequential composition. Rules, security functions, and chains transform flow traces, in particular through blocking certain traffic and modifying the values of certain header fields.

Definition 2. A security function s is a function from traffic (i.e., a sequence of packets) to traffic. For an integer $n \in \mathbb{N}$, the function $cut(t, n)$ returns the prefix of traffic t consisting of at most n packets. Given a predicate $pred(p)$ on packets, we define the function $restrict(t, pred)$ that returns the subsequence of traffic t consisting of those packets satisfying $pred$.
Security functions can be composed in sequence ($\circ_{\gg}$) or in parallel ($\circ_{+}$) where

$$(s_1 \circ_{\gg} s_2)(t) = s_2(s_1(t))$$
$$(s_1 \circ_{+} s_2)(t) = s_1(t) \ \oplus \ s_2(t)$$

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

4.2 Classifying flows for learning security requirements

Our first objective is to classify the flows observed for an application according to the attack types mentioned in section 3.1. As introduced in Sect. 3.4, the network trace t_{app} generated by an application is represented by a Markov chain whose locations L_{app} correspond to (not necessarily contiguous) sub-traces of t_{app} consisting of flows with the same *orgname* attribute. Transitions T_{app} of the Markov chain are triples (l, p, l') for locations $l, l' \in L_{app}$ and a probability value $p \in [0; 1]$. Observe that for any flow $f \in t_{app}$, there is a unique location $l \in L_{app}$ (corresponding to $f.orgname$) such that $f \in l$; we denote this location as l_f.
The analysis of the transition probabilities that occur in the Markov chain, and in particular those associated with self-loops, is at the basis of detecting potential attacks such as denial of service, port scanning or worm traffic.
We thus classify flows, and by extension their destination addresses, based on the following metrics defined for a flow trace t of length $n > 1$:

$$avg\_interval(t) = \frac{\sum_{i=2}^{n} t_i.timestamp - t_{i-1}.timestamp}{n-1}$$
$$avg\_size(t) = \frac{\sum_{i=1}^{n} t_i.packets}{n}$$
$$count(x, t) = |\{i \in 1\,..\,n : t_i.dstaddr = x \vee t_i.dstport = x\}|$$
$$ports(t) = \{p \in \textsc{Port} : \exists i \in 1\,..\,n : t_i.dstport = p\}$$
$$protocols(t) = \{p \in \textsc{Prot} : \exists i \in 1\,..\,n : t_i.protocol = p\}$$

In addition, *bgp_ranking*(ip) denotes a value corresponding to a trust ranking measure of the IP address ip. In practice, this value is obtained by contacting a remote service.
We associate the following thresholds to the above metrics; appropriate threshold values are defined by the network operators and administrators.

- *attack_limit*: maximum probability of self looping transitions for considering a location of the automaton as normal;
- *min_interval*: minimum acceptable interval time between the arrival of packets in a flow;
- *min_size*: minimum number of packets in a flow;
- *ip_limit*: maximum number of occurrences of an IP address;
- *port_limit*: maximum number of occurrences of a port number;
- *port_scan_limit*: minimum number of port numbers contacted in a trace for considering it as a port scanning trace;
- *unsafe_threshold*: maximum value of *bgp_ranking* for considering an IP address as safe.

At the core of our approach lies a classification of the destination addresses a appearing in flows in t_{app} according to the following predicates that indicate whether a is suspected to be the target of an attack of the various types we consider:

$$\begin{aligned}
dos(a) \;\equiv\;& f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge \\
& p \geq attack_limit \wedge count(a, l_f) \geq ip_limit \wedge \\
& avg_interval(l_f) \leq min_interval \wedge avg_size(l_f) \leq min_size \\
port_scan(a) \;\equiv\;& f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge \\
& p \geq attack_limit \wedge count(a, l_f) \geq ip_limit \wedge \\
& avg_interval(l_f) \leq min_interval \wedge avg_size(l_f) \leq min_size \wedge \\
& |\, ports(l_f) \,| \geq port_scan_limit
\end{aligned}$$

$$\begin{aligned}
worm(a, pt) \;\equiv\;& f \in t_{app} \wedge a = f.dstaddr \wedge (l_f, p, l_f) \in T_{app} \wedge \\
& p \geq attack_limit \wedge pt = f.dstport \wedge count(pt, l_f) \geq port_limit \\
botnet(a, pt) \;\equiv\;& f \in t_{app} \wedge a = f.dstaddr \wedge count(a, l_f) \geq ip_limit \wedge pt = f.dstport \wedge \\
& protocols(l_f) \cap \{\text{“tcp”}, \text{“udp”}\} \neq \emptyset \;\Rightarrow avg_interval(l_f) \leq min_interval \\
unsafe(a) \;\equiv\;& f \in t_{app} \wedge a = f.dstaddr \wedge bgp_ranking(a) \geq unsafe_threshold \\
safe(a) \;\equiv\;& \neg dos(a) \wedge \neg port_scan(a) \wedge \neg worm(a, pt) \wedge \neg botnet(a, pt) \wedge \neg unsafe(a) \\
danger(a, pm) \;\equiv\;& f \in t_{app} \wedge a = f.dstaddr \wedge pm \in P_{f.appname} \cap \mathcal{D}
\end{aligned}$$

In words, an address is considered to be the target of a potential attack if there exists a flow in t_{app} for which certain threshold values are exceeded. Addresses that are not the target of an attack are considered safe. In addition, the predicate *danger* records addresses that receive flows from an application that holds dangerous permissions. For example, a few properties derived for the Pokemon Go application, based on its automaton are given in Listing 1.1.

```
unsafe(169.45.223.20)
```

```
unsafe(37.58.73.183)
unsafe(54.241.184.32)
unsafe(54.241.165.61)
unsafe(173.192.233.91)
```

Listing 1.1. Example of addresses contacted by Pokemon Go that may be classified as unsafe.

4.3 Inferring a high-level representation of the chain

We now present a rule-based program for inferring the chain of security functions that should be deployed on the basis of the observed trace, making use of the classification of flows that occur in the trace. In a nutshell, we start by associating elementary security rules with addresses that occur in the trace. These will be composed in parallel to build security functions such as firewalls or intrusion detection systems, which in turn are composed sequentially to form the entire chain. In the present section, security functions are represented symbolically; we will explain in section 4.4 how we translate this representation into the Pyretic language.

The elementary security rules make use of the following predicates that are defined externally:

- $regexp(s, pm)$: true if the string s (representing the packet payload) matches a regular expression associated with the permission pm;
- $tcp\_check(t)$: true if the network traffic t is a valid TCP connection;
- $udp\_check(t)$: true if the network traffic t is a valid UDP connection;
- $http\_check(s)$: true if the string s (representing the packet payload) is a valid HTTP request;
- $inspect\_payload(s)$: true if the string s (representing the packet payload) passes deep packet inspection.

Our system is based on the following elementary security rules:

$$
\begin{aligned}
forward(a,t) &= restrict(t,\ \lambda pk : pk.dstaddr = a)\\
block(a,pt,t) &= restrict(t,\ \lambda pk : pk.dstaddr \neq a \wedge pk.dstport \neq pt)\\
limit(a,n,t) &= cut(forward(a,t), n)\\
filter(a,pm,t) &= restrict(t,\ \lambda pk : pk.dstaddr = a \wedge regexp(pk.payload, pm))\\
inspect(a,t) &= restrict(t,\ \lambda pk : pk.dstaddr = a \wedge inspect\_payload(pk.payload))\\
tcp(a,pt,t) &= \begin{cases} restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt) \\ \quad \text{if } tcp\_check(t) \\ \langle\rangle \text{ otherwise} \end{cases}\\
udp(a,pt,t) &= \begin{cases} restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt) \\ \quad \text{if } udp\_check(t) \\ \langle\rangle \text{ otherwise} \end{cases}\\
http(a,pt,t) &= restrict(t,\ \lambda pk : pk.dstaddr = a \wedge pk.dstport = pt \\
&\qquad\qquad \wedge\ http\_check(pk.payload))
\end{aligned}
$$

smShould $forward$ (and $deploy_{forward}$ below) take a port parameter, just like *block*?

The inference system below determines which security rules to deploy for addresses appearing in the given flow trace. For each of the elementary security rules r above, a corresponding predicate $deploy_r$ indicates if the rule is to be instantiated, with additional parameters corresponding to the relevant IP address, port etc.

$$
\begin{aligned}
deploy_{block}(a, pt) &\leftarrow worm(a, pt) \\
deploy_{block}(a, pt) &\leftarrow botnet(a, pt) \\
deploy_{forward}(a) &\leftarrow \neg worm(a, pt) \wedge \neg botnet(a, pt) \\
deploy_{limit}(a, ip\_limit) &\leftarrow dos(a) \\
deploy_{limit}(a, ip\_limit) &\leftarrow port\_scan(a) \\
deploy_{tcp}(a, pt) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge f.protocol = \text{“tcp”} \\
deploy_{udp}(a, pt) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge pt = f.dstport \wedge \\
&\quad\ pt \neq 80 \wedge pt \neq 443 \wedge f.protocol = \text{“udp”} \\
deploy_{http}(a, 80) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge f.dstport = 80 \\
deploy_{http}(a, 443) &\leftarrow f \in t_{app} \wedge a = f.dstaddr \wedge f.dstport = 443 \\
deploy_{filter}(a, pm) &\leftarrow unsafe(a) \wedge\ danger(a, pm) \\
deploy_{inspect}(a) &\leftarrow unsafe(a)
\end{aligned}
$$

Based on the predicates $deploy_r$ inferred to be true from a given trace t_{app} characterizing the network behavior of the application we wish to protect, we now construct security functions by composing elementary rules in parallel.

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

Continuing our example of the Pokemon Go application, we obtain a chain containing the following security functions. (We omit the full definitions since the inference system generates too many security rules, but show the overall structure of each security function.)

$$
\begin{aligned}
stateless\_firewall(t) &= forward(169.45.223.16, t) \circ_+ forward(169.45.223.20, t) \circ_+ \ldots \\
stateful\_firewall(t) &= tcp(169.45.223.16, 80, t) \circ_+ tcp(169.45.223.20, 80, t) \circ_+ \ldots \\
&\quad\ http(169.45.223.20, 80, t) \circ_+ \ldots \\
dpi(t) &= inspect(169.45.223.16, t) \circ_+ inspect(169.45.223.20, t) \circ_+ \ldots
\end{aligned}
$$

These security functions are in turn composed into chains to be applied to the network traffic of the different types:

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

These chains are deployed for filtering traffic generated by the target application by subjecting addresses to the chains associated with the classes to which the address belongs.
For the Pokemon Go application, we should deploy the chain corresponding to unsafe traffic. However, given that no dangerous permission is declared in the manifest file of this application, the DLP component of that chain is trivial and can be omitted in order to reduce the overall complexity of flow evaluation.

4.4 Generation of a Pyretic implementation of the chain

The last step of our approach consists in generating the Pyretic code implementing the abstract functions that we previously computed. Below we provide rewriting rules to derive Pyretic implementations corresponding to the elementary security rules introduced in section 4.3. In these rewrites, the argument of the security rules corresponding to the traffic t remains implicit in the Pyretic translation, which is applied to the current stream of packets. The functions *DPIQuery*, *TCPFilter*, *UDPFilter*, and *HTTPFilter* are part of our Synaptic checker [21] using dynamic query policies that Pyretic provides. The translations of the overall chains is then obtained by composing the Pyretic code sequentially or in parallel using the combinators $\gg$ and $+$ of Pyretic. The following definitions indicate the implementations of the elementary security rules;

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

To illustrate this last step of our inference system let us consider again our running example, the Pokemon Go application. The security functions introduced previously are converted into the following chain of security functions.

applications	# flows	# IP/ports	# sf	# rules
disneyland	282	5	4	44
dropbox	1000	17	5	311
faceswitch	151	30	5	425
lequipe	1000	208	4	1640
meteo	1000	89	4	716
ninegag	1000	124	4	930
pokemongo	275	24	5	485
ratp	779	3	4	28
skype	1000	442	5	6529
viber	1000	176	5	4163

Fig. 2. Network flows of Android applications considered during experiments, with the number of combinations of IP addresses and ports they contact, and the number of security functions (sf) and rules for the generated security chain.

$$
\begin{aligned}
\mathit{stateless\_firewall} &= \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.16}) + \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.20}) + \ldots \\
\mathit{stateful\_firewall} &= \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.16}, \mathit{dstport} = 80) \gg \mathit{TCPFilter} + \\
&\quad \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.20}, \mathit{dstport} = 80) \gg \mathit{TCPFilter} + \ldots + \\
&\quad \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.20}, \mathit{dstport} = 80) \gg \mathit{HTTPFilter} + \ldots \\
\mathit{dpi} &= \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.16}) \gg \mathit{DPIQuery} + \\
&\quad \mathit{match}(\mathit{dstaddr} = \mathit{169.45.223.20}) \gg \mathit{DPIQuery} + \ldots \\
\mathit{chain} &= \mathit{stateless\_firewall} \gg \mathit{stateful\_firewall} \gg \mathit{dpi}
\end{aligned}
$$

5 Performance evaluation

We implemented our inference system in SWI-Prolog (version 7.6.4). We evaluated the performances of the proposed solution through an extensive set of experiments. The experimental setup was based on a MacBookPro laptop with an Intel Core i7 (2.5 GHz) processor and 16 GB of RAM. During these experiments, we considered a set of log files (more than 7000 network flows) captured from Android applications summarized in Fig. 2.

These results clearly illustrate the high heterogeneity in the number of security functions and rules generated for each application. To minimize the impact of the deployment of several chains we designed a factorization algorithm presented in [20]: this algorithm allows us to group several chains of security functions into a larger one that contains only one occurence of each security function and at most as many security rules as the original chains.

In complement to these experiments we evaluated the accuracy of the different chains of security functions. We evaluated this metric by injecting a simple port scan of 50 flows in the log file of each application, and we quantify the accuracy as the ratio of the sum of true positive and true

Application	Avg. Acc.	Min. Acc.	Max. Acc.
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

Fig. 3. Accuracy of the chains generated for each application.

negative results over the total number of flows. Concretely, we used 70% of the logs for generating the chains and 30% for the evaluation. We also fixed a detection rate corresponding to the number of attack flows that must be matched before blocking the traffic and we varied this rate from 0 to 10; the corresponding results appear in Fig. 3 where we present the minimum, maximum and average accuracy measured for each chain.
We again observe a high heterogeneity in the results obtained for each application: for some applications, the accuracy is very close to 100% while others have a minimal accuracy lower than 50%. This is again caused by the nature of each log file: for some applications the 30% of logs used for evaluation only contain IP addresses that where already known during the learning phase and are therefore accepted by the chain while other applications have more disparities in their logs. The improvement of these results will require joint work with researchers working on attack detection to design more elaborated methods of detection. The approach proposed in this paper decouples this phase from the inference of chains through first order predicates, and this makes it easy to change the method for detecting attacks without modifying the general process.

6 Correctness properties of the generated chains

The construction of security chains based on a high-level representation guarantees certain correctness properties that we now discuss. Specific correctness properties of chains can be verified using formal verification techniques such as model checking or SMT solving, and our previous contribution [21] is intended for doing so; it is thus complementary to the work presented here.

6.1 Packet routing

Two elementary desirable properties of packet routing in networks are the absence of black holes and of loops. A black hole arises if packets

are sent to a link at which no actual routing function is installed. A loop refers to a cycle in routing policies, so that packets may be sent back to a security function that they have already passed. Our security functions avoid these problems by construction.

Lemma 1. *The security chains generated by the approach described in section 4 are free of black holes and of loops.*

Proof. In our setup, security functions are total functions on sequences of packets, and they are built up from elementary security rules using parallel and sequential composition. In particular, every constituent of our chains is fully defined before it is used, so black holes do not exist at the abstract level of the descriptions of the chains. Similarly, the high-level definitions of chains do not involve fixpoint operators or similar looping constructs. Finally, we rely on the close correspondence between the abstract chains and their Pyretic implementation and on the correctness of the Pyretic translator in order to ensure that the latter does not introduce black holes or loops. □

6.2 Shadowing freedom and coherence

The two main correctness properties of chains of security functions that we are interested in are *shadowing freedom* and *coherence*. Shadowing freedom means that whenever two rules are composed in parallel within a chain, only one of them actually applies. This property ensures that there is no confusion in the sense that two rules could be applied with potentially conflicting results. In particular, this property implies the *consistency* of the rules, which requires that whenever two rules apply, they result in the same decisions. Coherence means that the traffic after applying the security chains satisfies the security requirements: safe traffic passes unchanged, whereas potentially dangerous traffic is either blocked or limited within acceptable bounds.

Lemma 2. *The security chains generated by the approach described in section 4 guarantee shadowing freedom.*

Proof. Elementary security rules are composed in parallel in the definitions of the basic security functions *stateless_firewall*, *ids*, *stateful_firewall*, *dpi*, and *dlp*. The definition of *stateless_firewall* composes in parallel rules *forward* and *block*, which are potentially in contradiction. However, this is possible only if both $deploy_{forward}(a)$ and $deploy_{block}(a, pt)$ are true for some address a and port pt, and this is impossible due to the definitions of these predicates.
Similarly, the parallel composition of the elementary security rules *tcp*, *udp*, and *http* in the definition of *stateful_firewall* is unproblematic because the definitions of the corresponding predicates $deploy_{tcp}$, $deploy_{udp}$, and $deploy_{http}$ are mutually exclusive. □

We now show that our security chains are coherent with the security policy determined on the basis of the trace t_{app} underlying their generation.

Lemma 3. *Given a trace t_{app} characterizing the network traffic generated by an application, the security chains generated by the approach described in section 4 transmit unchanged the traffic towards addresses considered as safe but block or limit network traffic towards other addresses.*

Proof. An address is considered as potentially not safe if t_{app} contains some flow towards that adress classified as worm, botnet, dos, port scanning or unsafe. The stateless firewall applied as the first security function in the chain will directly block packets towards IP addresses associated with worm and botnet traffic.
Concerning traffic directed to addresses associated with DoS or port scanning attacks, it will pass the stateless firewall and will subsequently be transmitted to the IDS. The traffic will then be limited to a number of packets bounded by the fixed threshold *ip_limit*, considered to be acceptable by the security policy.
For addresses associated with unsafe flows, i.e., network traffic potentially compromising sensitive data, the security chains contain the security functions DPI and DLP that check the payload of packets. These apply the security policy by blocking packets that do not match the criteria defined by the predicates *regexp* (associated with Android permissions) and *inspect_payload*.
Traffic towards addresses classified as safe is only subject to the stateless firewall, which lets it pass unchanged. □

7 Conclusions and future work

We proposed in this paper a declarative approach based on inference rules for automating the generation of chains of security functions, based on the requirements of end users. This inference system is intended to protect Android applications, by taking into account both their networking behavior and the OS-level permissions that they request. By using first-order predicates for classifying network traffic observed in the flow trace – rather than for example finite state machines – the composition and factorization of security chains to be applied for several applications becomes straightforward. Our system infers a high-level representation of the security functions, which can be translated into a concrete implementation in the Pyretic language for programming software-defined networks. We showed that the generated chains satisfy several desirable properties such as the absence of black holes and of loops, shadowing freedom, and that they are consistent with the underlying security policy.
Further correctness properties of the chains can be verified using our Synaptic checker [21] based on symbolic model checking and SMT solving. The main assumption underlying our approach is that the network-level behavior of Android applications can be characterized in terms of

flow traces that are collected before the security chains are generated, and that are analyzed offline, as described in our previous work [22] on process learning. This assumption holds for many, but not all Android applications, a Web browser being a typical counter-example. In that case, network administrators must install a default security chain.
An interesting extension of our present work would be to consider which parts of the analysis are sufficiently lightweight to be performed online. As further perspectives, we are also planning to work on optimizing and improving the parameterization of the security chains that are generated by our inference system. In addition, we are interested in investigating to what extent our solution is compatible with network function virtualization techniques (NFV) to implement security functions, such as firewalls and intrusion detection systems.

References

1. Ain, A., Bhuyan, M.H., Bhattacharyya, D.K., Kalita, J.K.: Rank correlation for low-rate ddos attack detection: An empirical evaluation. I. J. Network Security **18**(3), 474–480 (2016)
2. Al-Shaer, E.S., Hamed, H.H.: Discovery of policy anomalies in distributed firewalls. In: Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications (INFOCOM 2004) (2004)
3. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: Vericon: Towards Verifying Controller Programs in Software-Defined Networks. In: Proc. 35th ACM SIGPLAN Intl. Conf. Programming Language Design (PLDI'14). pp. 282–293. Edinburgh, UK (2014)
4. Barthel, D., Vasseur, J., Pister, K., Kim, M., Dejean, N.: Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks. RFC 6551 (Mar 2012). https://doi.org/10.17487/RFC6551
5. Biere, A., Heule, M., van Maaren, H., Walsch, T.: Handbook of satisfiability. IO press (2008)
6. Chen, X., Heidemann, J.: Detecting early worm propagation through packet matching. Tech. Rep. ISI-TR-2004-585, USC/Information Sciences Institute (Feb 2004), `http://www.isi.edu/%7ejohnh/PAPERS/Chen04a.html`
7. Claise, B.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. RFC 5101 (Jan 2008). https://doi.org/10.17487/RFC5101
8. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking, LNCS, vol. 5000, pp. 1–26. Springer (2008)
9. Ensafi, R., Park, J.C., Kapur, D., Crandall, J.R.: Idle port scanning and non-interference analysis of network protocol stacks using model checking. In: Proceedings of the 19th USENIX Conference on Security. pp. 17–17. USENIX Security'10, USENIX Association, Berkeley, CA, USA (2010)
10. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: Bernardo, M.,

Issarny, V. (eds.) Formal Methods for Eternal Networked Software Systems (SFM'11). LNCS, vol. 6659, pp. 53–113. Springer, Bertinoro, Italy (2011)
11. Foster, N., Freedman, M.J., Guha, A., Harrison, R., Kata, N.P., Monsanto, C., Reich, J., Reitblatt, M., Jennifer, R., Schlesinger, C., Story, A., Walker, D.: Languages for Software-Defined Networks. In: Software Technology Group (2016)
12. Foster, N., Freedman, M.J., Harrison, R., Monsanto, C., Walker, D.: Frenetic, a Network Programming Language. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11) (2011)
13. García, S., Grill, M., Stiborek, J., Zunino, A.: An empirical comparison of botnet detection methods. Comput. Secur. **45**, 100–123 (Sep 2014). https://doi.org/10.1016/j.cose.2014.05.011
14. Handley, M.J., Rescorla, E., Internet Architecture Board: Internet Denial-of-Service Considerations. RFC 4732 (Dec 2006). https://doi.org/10.17487/RFC4732
15. Hurel, G., Badonnel, R., Lahmadi, A., Festor, O.: 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'15) (2015)
16. Hurel, G., Badonnel, R., Lahmadi, A., Festor, O.: Towards Cloud Based Compositions of Security Functions for Mobile Devices. In: IFIP/IEEE International Symposium on Integrated Network Management (IM'15) (2015)
17. Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., Clark, R.: Kinetic: Verifiable Dynamic Network Control. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15) (2015)
18. Lahmadi, A., Beck, F., Finickel, E., Festor, O.: A platform for the analysis and visualization of network flow data of android environments. IFIP/IEEE International Symposium on Integrated Network Management (IM) (May 2015). https://doi.org/10.1109/INM.2015.7140443, poster
19. Malkin, G.S., Parker, T.L.: Internet Users' Glossary. RFC 1392 (Jan 1993). https://doi.org/10.17487/RFC1392
20. Schnepf, N., Badonnel, R., Lahmadi, A., Merz, S.: Automated factorization of security chains in software-defined networks (2018), submitted for publication
21. Schnepf, N., Merz, S., Badonnel, R., Lahmadi, A.: Automated verification of security chains in software-defined networks with Synaptic. In: Proceedings of the 3rd IEEE Conference on Network Softwarization (NetSoft'17) (2017)
22. Schnepf, N., Merz, S., Badonnel, R., Lahmadi, A.: 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'18) (2018)
23. Sperotto, A.: Flow-based intrusion detection. Ph.D. thesis, University of Twente (2010). https://doi.org/10.3990/1.9789036530897