Automated Factorization of Security Chains in Software-Defined Networks

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, and Stephan Merz
Université de Lorraine, CNRS, Inria, LORIA
Campus Scientifique, Villers-les-Nancy, France
{schnepf, badonnel, lahmadi, merz}@inria.fr

***Abstract*—Software-defined networking (SDN) offers new perspectives with respect to the programmability of networks and services. In particular in the area of security management, it may serve as a support for building and deploying security chains in order to protect devices that may have limited resources. These security chains are typically composed of different security functions, such as firewalls, intrusion detection systems, or data leakage prevention mechanisms. In previous work, we suggested the use of techniques for learning automata as a basis for generating security chains. However, the complexity and the high number of these chains induce significant deployment and orchestration costs. In this paper, we propose and evaluate algorithms for merging and simplifying these security chains in software-defined networks, while keeping acceptable accuracy. We first describe the overall system supporting the generation and factorization of the security chains. We then present the different algorithms supporting their merging, and finally we evaluate the solution through an extensive set of experiments.**

***Index Terms*—Security Management, Software-Defined Networking, Chain Synthesis, Merging Methods**

I. Introduction

The interest in software-defined networking (SDN) is mainly due to its facilities for programming and managing networks and services [1], [2]. The SDN paradigm decouples networks into a data plane and a control plane. The data plane typically corresponds to programmable switches deployed in the network, while the control plane supports the configuration of these switches by one or several controllers that can react to network events and adapt the network configuration to specific contexts. The south-bound interface relies on dedicated protocols, such as the OpenFlow protocol, while the north-bound interface benefits from higher level languages, such as the Pyretic language [7]. This language, part of the Frenetic framework [8] and embedded in Python, is intended for configuring the controller and describing the forwarding strategies. A Pyretic program is compiled into low-level rules that are interpretable by switches.

In the area of security management, this programmability brings flexibility to the deployment and adjustment of security mechanisms in network infrastructures. In particular, it enables dynamically building and orchestrating chains that compose different security functions. For instance, these security functions may include firewalls, intrusion detection systems, and data leakage prevention mechanisms. They may directly be implemented on the SDN layer or be provided as middleboxes using network function virtualization. They can be used to observe and protect applications on devices that may have restricted resources, such as mobile devices and connected objects. While it is possible to automatically build security chains that are consistent and satisfy security properties, it may generate numerous and complex security chains and result in several thousands of rules even for simple configurations. In order to reduce the complexity of security chains that are to be deployed on an SDN platform, we propose in this paper an automated method for factorizing security chains. The objective is to support efficient merging and simplification of a set of security chains while still providing adequate protection to applications. Our main contributions in this paper are: (i) designing the overall architecture supporting the factorization of security chains in software-defined networks, (ii) specifying dedicated algorithms for factorizing a set of security chains and their security functions, (iii) implementing a proof-of-concept prototype in Python and Prolog, and (iv) evaluating its performances through experiments carried out in Android environments.

The remainder of this paper is organized as follows. Section II gives an overview of existing work related to security chains. Section III provides some important background with respect to the model and generation of security chains in our context. Section IV describes the proposed factorization approach, through the description of the considered architecture and the specification of algorithms. Section V gives a performance evaluation of our solution. Section VI concludes the paper and points out future research perspectives.

II. Related work

Constructing security chains is an important topic that has been covered quite extensively in the literature. Hurel et al. [3], [4] have shown the benefits of building such chains that can be partially deployed on cloud or fog infrastructures, to support the protection of end-user devices. Techniques for chaining service functions such as [5] support the composition of virtualized network functions, including security functions, and benefit from deployment optimization techniques. Another approach consists in synthesizing security chains depending on intended properties: in [6] we proposed a logical system based on constraint programming for the automated generation of security chains. Nevertheless these approaches do not

978-3-903176-15-7 © 2019 IFIP

address the case where several chains have to be generated or orchestrated to gather in a virtual environment.

There also exists interesting work on high level languages dedicated to programming SDN controllers with support for verification. In particular, the Pyretic language, part of the Frenetic family of SDN programming languages [7], [8], provides a simple and intuitive syntax for specifying security policies at the level of a SDN controller. It includes an extension, called Kinetic [9], for the verification of the control plane in software-defined networks. We previously introduced a complementary technique [10] for verifying both the control and the data plane. These techniques aim at verifying security chains a posteriori, while our current work aims at methods that build security chains that ensure certain properties a priori.

Complementary to formal verification the placement of security chains depending on network resources is also a challenging research problem. In [11] authors proposed an interesting approach for the efficient provisioning of security chains in SDN; nevertheless this work only concerns placement of single chains and do not consider the case where several chains have to be placed. In [12] authors proposed another interesting approach for provisioning security functions based on application requirements; nevertheless this work again focus on optimizing single chains, placing large sets of chains remains a lacking problem in the literature.

III. Background on security chains

Before we present our factorization algorithms, we explain how we represent security chains and introduce relevant notation. A security chain c corresponds to a graph whose vertices are security functions, such as firewalls, intrusion detection systems (IDS) and data leakage prevention services (DLP) that are applied to the network traffic [4]. In the following, we assume for simplicity that a given chain contains at most one security function of any given type, and our algorithms preserve this property. This assumption is however not fundamental to our approach. A chain c is characterized by the two attributes $c.secFunctions$, representing the set of security functions it contains, and $c.edges$ representing the links between these security functions. The operation $c.getTypes()$ retrieves the types of security functions in c, while $c.getSecFunction(tp)$ returns the (unique) security function of the chain corresponding to a given type tp.

Each security function sf is characterized by three attributes: $sf.type$, $sf.rules$ and $sf.default$. The $sf.type$ attribute indicates the type of the security function, such as stateless firewall. The $sf.rules$ attribute contains a dictionary d of rules. Each rule r has two attributes $r.guard$ and $r.action$. The actions are indexed based on their corresponding guard in the dictionary. We can retrieve the set of keys of the dictionary with the $d.getKeys()$ operation. Finally, the $sf.default$ attribute defines a default action to be applied, when there is no guard that matches the traffic. The behavior of a security function is as follows: incoming network traffic is matched against the rules of the security function. If some guard matches the traffic, then the security function applies the corresponding action. If no guard matches, it applies the default action. When multiple guards match, one of the corresponding actions is chosen according to a prioritization based on the index in the dictionary. Each edge e is characterized by two attributes corresponding to its source $e.src$ and its destination $e.dst$ to interconnect security functions.

The security functions are modeled with the Pyretic language for the sake of compatibility with our previous work, nevertheless they can be supported by moderner SDN programming languages. Security chains can be composed in sequence and in parallel. These operations may also be applied to individual security functions, considered as singleton chains. The sequential composition $c_1 \gg c_2$ of two security chains c_1 and c_2 leads to a composite security chain where all packets accepted by c_1 are then forwarded to c_2. Similarly, the parallel composition operation $c_1 + c_2$ leads to a composite security chain where all packets are analyzed in parallel by c_1 and c_2. Security chains are instantiated for software defined networks supporting the Pyretic language. The security functions that compose them are directly implemented on SDN switches that enable packet forwarding, blocking, and counting. Advanced analysis techniques such as deep packet inspection are obtained by forwarding packets to the controller, which is natively supported by the Pyretic framework.

A. Baseline approaches for the combination of chains

In the context where several applications would be protected in parallel we would have several security chains requiring to be combined. Before introducing our factorization algorithm we will point out two approaches that serve as baselines.

A first naive approach for combining chains consists in composing a set of security chains in parallel using the composition operator $+$ provided by Pyretic. We refer to this approach as the parallel merging of chains. Whereas its implementation is simple, this solution generates a number of security functions that grows linearly with respect to the number of applications to be protected. The number of security functions as well as security rules is kept unchanged with respect to the initial set of security chains. The only benefit is that it leads to a single security chain to be deployed.

A second approach, that we call grouped merging, consists in applying the behavior learner and inference engine simultaneously for a set of applications. Given a fixed set of types of security functions, this approach bounds the number of functions that will be generated, since we generate at most one function per type. However, it introduces additional complexity with respect to some security functions, such as data leakage prevention, which combines properties related to network traces with the permissions associated with applications. The corresponding rules are therefore generated based on the Cartesian product of the unsafe IP destination addresses to be matched and the data to be protected. When multiple IP addresses and multiple permissions need to be considered, this leads to a quadratic number of rules. Moreover, the Cartesian product changes the semantics of the generated security chain compared to individual chains. Instead of associating the unsafe addresses of an application with the corresponding permissions, each unsafe IP address is associated with each permission, which may result in an overly restrictive chain.

IV. Automated factorization of security chains

We described a technique for inferring security chains based on learning finite automata (Markov models) that represent the network behavior of an Android application, in previous work [13]. This approach tends to generate a high number of security chains, and we therefore propose algorithms for combining a set of chains into a single synthetic chain. Our algorithms can however be useful for merging chains that are constructed in a different way. We will briefly describe the system that we propose for constructing chains and then present the factorization algorithms. Our case study is focused on Android applications, but the overall ideas apply to other networked environments as well.

A. Overview of the considered system

The overall system that we propose is driven by an orchestrator interacting with different components to analyze the behavior of applications, generate adequate security chains and factorize them into a single security chain. This security chain is then deployed by a controller in the SDN environment using programmable switches. The main components are the orchestrator driving the system, the learner building the behavioral model of applications, the generator inferring security chains, the storage manager storing security chains, and the factorizer that merges security chains.

$$\begin{aligned}
\mathit{stl\_fw} &= R_{(1,1)} + R_{(1,2)} + R_{(1,3)} + \ldots + R_{(1,n_1)} \\
\mathit{ids} &= R_{2,1)} + R_{(2,2)} + R_{(2,3)} + \ldots + R_{(2,n_2)} \\
\mathit{stf\_fw} &= R_{(3,1)} + R_{(3,2)} + R_{(3,3)} + \ldots + R_{(3,n_3)} \\
&\quad +\mathit{match}(\mathit{dstip} = \mathit{45.67.89.123}) \gg \\
&\quad (\mathit{TCPFilter} + \mathit{HTTPFilter}) \\
\mathit{dlp} &= R_{(4,1)} + R_{(4,2)} + R_{(4,3)} + \ldots + R_{(4,n_4)} \\
\mathit{dpi} &= R_{(5,1)} + R_{(5,2)} + R_{(5,3)} + \ldots + R_{(5,n_5)} \\
\mathit{chain} &= \mathit{stl\_fw} \gg \mathit{ids} \gg \mathit{stf\_fw} \gg \mathit{dlp} \gg \mathit{dpi}
\end{aligned}$$

Figure 1. Example of a security chain generated for protecting a given Android application.

The behavior learner receives network traces that are produced by Android applications, captured using dedicated network probes. It builds finite automata characterizing the applications to be protected. This analysis can be performed offline, assuming that application behaviors are reasonably stable. The chain generator interprets these models, determines their properties, and automatically generates corresponding security chains, relying on logical inference rules. Since security chains are inferred for each model separately, a large number of chains may be generated, and each chain may contain many rules. An example of a simple chain generated by the behavior learner to protect an application is given in Fig. 1 where $R_{(i,j)}$ denotes rules indexed on the index of the security function and of the rule. This security chain is composed of five security functions, a stateless firewall noted $\mathit{stl\_fw}$, an intrusion detection system noted ids, a stateful firewall noted $\mathit{stf\_fw}$, a data leakage prevention service noted dlp and a deep packet inspection service noted dpi. Each security function is specified by indicating its rules. In this simple example, the overall chain is obtained by composing the security functions in sequence. The performance bottleneck is the behavior learner: it can take minutes to elaborate the finite automata of an application. After generation, security chains are stored in a database, where they are indexed based on the name of the application for which they were generated. Thus, learning and inference are decoupled from the factorization and deployment of security chains, and chains can be looked up efficiently at run time.

B. Factorizing security chains

The objective of factorizing security chains is to transform a set of security chains into a single one. It aims at minimizing the overall number of security functions that are involved in the protection of an Android device and its applications, but also to reduce the number of rules that are required to define these security functions. In contrast with the two baseline approaches presented in the previous section, our merging approach consists in factorizing a set of security chains, after having generated them. It is based on two algorithms: $\mathit{merge\_functions}$ (Algorithm 1) produces a single security function from two given functions, while $\mathit{merge\_chains}$ (Algorithm 2) factorizes two entire security chains. In order to handle potential conflicts when combining two functions that may have contradictory rules for certain network traffic, we assume given priorities amongst rules, abstractly represented by an operator $\leq_r$ where smaller rules have higher priority. Different realizations of $\leq_r$ can be implemented by the network administrator, such as taking into account the level of reliability of the IP destination addresses.

Algorithm 1 takes two security functions sf_1 and sf_2 (assumed to be of the same type) as inputs. It first retrieves the sets of guards of sf_1 and sf_2 and evaluates their intersection and symmetric difference. Rules whose guards do not have a counterpart in the other function cannot be in conflict. They are are therefore simply added to the resulting function. For guards that appear in both security functions, the priorization operator $\leq_r$ is used to decide which action should be associated to the guard in the resulting security function. Algorithm 2 merges two security chains c_1 and c_2 and relies on the previous algorithm. It first identifies the security functions that have the same type in both security chains. It applies Algorithm 1 in order to merge these functions. Security functions of a type that has no counterpart in the other chain are simply added to the resulting chain. Finally, the edges of the resulting chain are built based on those of the considered security functions described in the input chains. By repeatedly applying Algorithm 2 to the chains generated for a set of Android applications, we obtain a single chain for protecting all applications.

Algorithm 1 Factorization of security functions.

```
function MERGE_FUNCTIONS($sf_1, sf_2 : SecFunction$)
                ▷ Build the rules appearing in only one function
    $guards_1 := sf_1.getKeys()$
    $guards_2 := sf_2.getKeys()$
    $rules := new\ Dictionary()$
    for $g \in guards_1 \setminus guards_2$ do
        $rules.put(g, sf_1.rules.get(g))$
    end for
    for $g \in guards_2 \setminus guards_1$ do
        $rules.put(g, sf_2.rules.get(g))$
    end for
                ▷ Add the rules matching the same network traffic
    for $g \in guards_1 \cap guards_2$ do
        $r_1 := sf_1.rules.get(g)$
        $r_2 := sf_2.rules.get(g)$
        if $r_1 \leq_r r_2$ then
            $rules.put(g, r_1)$
        else
            $rules.put(g, r_2)$
        end if
    end for
    return $new\ SecFunction(sf_1.type, rules)$
end function
```

Algorithm 2 Factorization of security chains.

```
function MERGE_CHAINS($c_1, c_2 : SecChain$)
    $res$ := new $SecChain()$
                ▷ Factorize security functions having the same type
    for $sf_1 \in c_1.secFunctions, sf_2 \in c_2.secFunctions$ do
        if $sf_1.type = sf_2.type$ then
            $sf$ := MERGE_FUNCTIONS($sf_1, sf_2$)
            $res.secFunctions.add(sf)$
        end if
    end for
                ▷ Add remaining functions to the result
    for $sf \in c_1.secFunctions \setminus res.secFunctions$ do
        if $sf.type \notin c_2.getTypes()$ then
            $res.secFunctions.add(sf)$
        end if
    end for
    for $sf \in c_2.secFunctions \setminus res.secFunctions$ do
        if $sf.type \notin c_1.getTypes()$ then
            $res.secFunctions.add(sf)$
        end if
    end for
                ▷ Add the proper connecting edges to the result
    for $cn \in c_1.edges$ do
        $src := res.getSecFunction(cn.src.type)$
        $dst := res.getSecFunction(cn.dst.type)$
        $cn := new\ Edge(src, dst)$
        $res.edges.add(cn)$
    end for
    for $cn \in c_2.edges$ do
        $src := res.getSecFunction(cn.src.type)$
        $dst := res.getSecFunction(cn.dst.type)$
        $cn := new\ Edge(src, dst)$
        $res.edges.add(cn)$
    end for
    return $res$
end function
```

C. Examples of merged security chains

We will illustrate the chains obtained using the different approaches discussed previously, using a simple example. Consider three Android applications app_1, app_2 and app_3. The behavior learner builds Markov models for each application based on the traces collected by network probes and on the permissions requested by each application. From these models, the chain generator infers a security chain, expressed in Pyretic, for each application. The example shown in Fig. 1 corresponds to the chain generated for protecting the first application app_1. Similar security chains are generated for protecting the applications app_2 and app_3. For simplicity, we will focus on the security rules related to the data leakage prevention (DLP) security functions that these security chains contain. We will denotes the guards of security rules by G_i and the corresponding actions by A_j but these abstract model of rules can be implemented in Pyretic or in any other SDN programming language. Parallel merging will result in a single security chain, part of which appears in Fig. 2. It is formed by composing in parallel three DPL security functions, noted dlp_1, dlp_2, and dlp_3. The security rules associated to each security function are unchanged from the initial set of security chains. In this example, we obtain a chain containing 3 security functions and 10 security rules.[1] As we explained previously, grouped merging involves building the Cartesian product of IP addresses and permissions for constructing the combined chain. In our example, assuming that IP addresses and permissions of the considered applications are distinct, we obtain a single DPL security function containing 18 security rules. In contrast, merging the chains using Algorithm 2 factorizes both security functions and their constituent rules. The DLP security function of the resulting chain is shown in Fig. 3: we obtain a single function containing 9 rules, which is smaller than the results of the two other approaches.

$$\begin{aligned}
dlp_1 &= G_1) \gg A_1) + G_2) \gg A_1) \\
dlp_2 &= G_1 \gg A_1) + G_1) \gg A_2 + \\
&\quad\; G_3) \gg A_1) + G_3) \gg A_2) + \\
&\quad\; G_4) \gg A_1) + G_4) \gg A_2) \\
dlp_3 &= G_5) \gg A_3) + G_6) \gg A_3)
\end{aligned}$$

Figure 2. Extract of a security chain resulting from parallel merging.

[1]Observe that parallel merging will typically lead to chains that contain several functions of the same type.

$$\begin{aligned} dlp \quad = \quad & G_1) \gg A_1) + G_2) \gg A_1) + \\ & G_1) \gg A_2) + G_3) \gg A_1) + \\ & G_3) \gg A_2) + G_4) \gg A_1) + \\ & G_4) \gg A_2) + G_5) \gg A_3) + \\ & G_6) \gg A_3) \end{aligned}$$

Figure 3. Extract of a security chain obtained using factorized merging.

V. Performance Evaluation

The proposed system has been implemented in Python. It includes a total of 8869 lines of code, including the implementation of the three merging approaches. The chain generator implemented in SWI-Prolog (version 7.6.4) is based on the logical inference rules described in [6] for generating the initial security chains. 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 Table I. In order to compare the performances of the merging approaches, we considered the following evaluation criteria:

- the complexity of the resulting security chains, measured as the numbers of security functions and rules;
- the overhead induced by the proposed system in terms of response times;
- the detection accuracy of the security chains.

A. Complexity of security chains

In a first series of experiments we are interested in evaluating the complexity of security chains, with respect to the number of security functions and rules. We compared the three different approaches: parallel merging, which simply composes chains in parallel after their generation, grouped merging, which directly generates a security chain for a set of applications to be protected, and finally factorized merging where the chains are invidually generated for each application and are then combined using the proposed algorithms. The obtained results with respect to the number of security rules are synthesized in Table II, plotting the total number of security rules for the three different approaches. The curves corresponding to the parallel and factorized merging turned out to be identical during these experiments, although they may differ in general, depending on the level of redundancy amongst rules. As expected, we can observe that the number of security rules for the first and third approaches (parallel and factorized merging approaches) grows linearly as a function of the number of applications to be protected, while this number grows quadratically with the second approach (grouped merging approach). As previously mentioned, this phenomenon can be explained by the fact that some security rules, such as the ones related to data leakage prevention, rely on the Cartesian product of parameters, such as IP addresses and application permissions. The number of security functions (not shown in the figure) grows linearly with the parallel merging but is constant with the grouped and factorized merging approaches. These experiments clearly show the benefits of the factorization algorithms for minimizing the numbers of security functions and security rules in the security chain.

Table I
ANDROID APPLICATIONS CONSIDERED DURING EXPERIMENTS, WITH STATISTICS ON THEIR FLOWS, SECURITY FUNCTIONS (SF) AND RULES.

App.	# 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

Table II
NUMBER OF RULES OF THE DIFFERENT SECURITY CHAINS.

App.	Parallel merging	Grouped merging	Factorized 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

B. Overhead of merging chains

In a second series of experiments we quantify the overhead of the proposed system in terms of response time. The objective is to evaluate the feasibility of the proposed solution in practice. These results do not include the learning phase related to the building of Markov models, but they include the time taken by the inference engine. The response times of the three different generation methods are not presented here, nevertheless we can clearly observe that the overhead of factorized merging compared to naive parallel merging is negligible. Grouped merging provides better performances than either parallel or factorized merging, this is due to the fact that there is only one chain generated for the entire set of applications. The downside to grouped merging is that is also requires an execution of the learning phase just before the chain inference. Assuming that chains for individual applications can be retrieved from the database, a more realistic comparison of the response times of grouped and factorized merging appears in Table III: the overall time including the additional

Table III
OVERALL TIME FOR GROUPED AND FACTORIZED MERGING.

App.	With learning time	With database retrieval
1	32.798	0.473
2	421.186	1.042
3	473.458	1.484
4	483.122	1.924
5	635.582	2.413
6	641.354	2.863
7	999.467	3.557
8	1181.953	4.189
9	1226.743	4.634
10	1421.65	5.139

learning phase of grouped merging dominates that required by factorization by more than two orders of magnitude.

These results clearly illustrate the fact that learning Markov models at runtime is not feasible. In our overall architecture, we suggest that Markov models and security chains for individual applications be computed proactively and that the security chains be stored in a database. Depending on the applications that are active on the device, the corresponding security chains are loaded and merged into a single chain that is then deployed by the SDN controller. Assuming that applications are stable over a relatively long period, the cost of learning can thus be amortized over several phases of deployment.

C. Accuracy of the security chains

Our third series of experiments addresses the accuracy of the security chains with respect to detecting attacks or misbehavior. We quantify accuracy using the numbers of true/false positives and true/false negatives observed for the security chains. More precisely, we measure accuracy as the ratio between the sum of true positives and true negatives vs. the number of flows. During the experiments, we considered the case of a simple port scanning attack of 50 flows. We used the first 70% of the application logs for the learning phase, while the remaining 30% was used for the evaluation of accuracy. To evaluate the accuracy of the individual chains, we fixed a certain detection rate, defined as the number of flows corresponding to an attack that must be observed before blocking the traffic to the concerned IP address. We varied this detection rate in order to measure the average, minimum and maximum values of accuracy, that are synthesized in the first, third and fourth columns on Table. IV. We can observe that this accuracy may highly differ from one application to another, due to the profile of applications.

We wanted also to quantify whether the merging approaches had an impact on this accuracy. We performed the same experiments, but considering each of the three security chains produced by the different merging approaches (parallel, grouped and factorized). We expected that the grouped merging approach induced degraded accuracy performances. This was not the case: the performance results were equal for the three merging approaches with our data set, and were identical to the performances provided by the security chains taken individually. We expected this to be the case for the parallel and factorized merging approaches which largely preserve the behavior of the individual security chains. The obtained results may be explained by the fact that applications mostly contact distinct IP addresses, which minimizes potential conflicts with respect to security rules. The average accuracy values for each of the three merged security chains are represented by a single column (second column) in Table IV. This one clearly shows that the merging approaches preserved accuracy performances. The same phenomenon was observed for the minimum and maximum accuracy values.

Table IV
ACCURACY OF INDIVIDUAL AND MERGED SECURITY CHAINS WITH RESPECT TO EACH APPLICATION TO BE PROTECTED.

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

VI. CONCLUSIONS AND FUTURE WORK

We have proposed in this paper an automated method for factorizing security chains in software-defined networks. This method is intended as a complement to the inference-based generation techniques we proposed in earlier work. The factorization algorithms presented here are designed to compose several security chains into a single one in order to minimize the number of security functions and rules while preserving the semantics of the original chains. The presented algorithms have been implemented in Python and have been integrated into our proof-of-concept prototype that also contains the learning and inference components. The performance of this implementation has been evaluated through a series of experiments. In particular, we have compared different approaches to factorizing security chains in terms of the complexity of the resulting chains, their accuracy, and the overhead incurred in computing the combined chains. The proposed factorization method is able to minimize the number of security functions and rules. It also facilitates the building of security chains at runtime, through a decoupling from the generation of individual chains.

As future work, we are interested in performing complementary experiments with more elaborated attacks, based on additional datasets collected from Android applications, but also other constrained environments, such as connected objects. We are also planning to pursue our efforts on formal verification and to investigate different optimization techniques for supporting the deployment of these security chains.

References

[1] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN, an Intellectual History of Programmable Networks," *SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[2] N. Feamster and H. Kim, "Software-Defined Networks: Improving Network Management with SDN," in *IEEE Communications Magazine*, February 2013.

[3] G. Hurel, R. Badonnel, A. Lahmadi, and O. Festor, "Towards Cloud Based Compositions of Security Functions for Mobile Devices," in *IFIP/IEEE International Symposium on Integrated Network Management (IM'15)*, 2015.

[4] ——, "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.

[5] A. F. Ocampo, J. Gil-Herrera, P. H. Isolani, M. C. Neves, J. F. Botero, S. Latré, L. Zambenedetti, M. P. Barcellos, and L. 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)*. Springer International Publishing, 2017, pp. 62–76.

[6] N. Schnepf, S. Merz, R. Badonnel, and A. Lahmadi, "Rule-Based Synthesis of Chains of Security Functions for Software-Defined Networks," in *Proceedings of the 18th International Workshop on Automated Verification of Critical Systems (AVOCS'18)*, 2018.

[7] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Kata, C. Monsanto, J. Reich, M. Reitblatt, R. Jennifer, C. Schlesinger, A. Story, and D. Walker, "Languages for Software-Defined Networks," in *Software Technology Group*, 2016.

[8] N. Foster, M. J. Freedman, R. Harrison, C. Monsanto, and D. Walker, "Frenetic, a Network Programming Language," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP'11)*, 2011.

[9] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable Dynamic Network Control," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[10] N. Schnepf, S. Merz, R. Badonnel, and A. Lahmadi, "Automated Verification of Security Chains in Software-Defined Networks with Synaptic," in *Proceedings of the 3rd IEEE Conference on Network Softwarization (NetSoft'17)*, 2017.

[11] A. S. Sendi, Y. Jarraya, M. Pourzandi, and M. Cheriet, "Efficient provisioning of security service function chaining using network security defense patterns," *IEEE Transactions on Services Computing*, p. 1, 2017. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TSC.2016.2616867

[12] R. Doriguzzi-Corin, S. Scott-Hayward, D. Siracusa, and E. Salvadori, "Application-centric provisioning of virtual security network functions," in *Proceedings of the 3rd IEEE Conference on Network Functions Virtualization and Software Defined Networking (IEEE NFV-SDN 2017)*, 2017.

[13] N. Schnepf, S. Merz, R. Badonnel, and A. 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'18)*, 2018.