Automated Verification of Security Chains in Software-Defined Networks with Synaptic

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, and Stephan Merz
Université de Lorraine, CNRS, Inria, Loria, F-54000 Nancy

ABSTRACT

Software-defined networks provide new facilities for deploying security mechanisms dynamically. In particular, it is possible to build and adjust security chains to protect the infrastructures, by combining different security functions, such as firewalls, intrusion detection systems and services for preventing data leakage. It is important to ensure that these security chains, in view of their complexity and dynamics, are consistent and do not include security violations. We propose in this paper an automated strategy for supporting the verification of security chains in software-defined networks. It relies on an architecture integrating formal verification methods for checking both the control and data planes of these chains, before their deployment. We describe algorithms for translating specifications of security chains into formal models that can then be verified by Satisfiability Modulo Theories (SMT) solving or model checking. Our solution is prototyped as a package, named Synaptic, built as an extension of the Frenetic family of SDN programming languages. The performances of our approach are evaluated through extensive experimentations based on the CVC4, veriT, and nuXmv checkers.

Keywords: Security Management, Software-Defined Networking, Formal Verification.

I. INTRODUCTION

The growing dynamics and size of the Internet pose new challenges in terms of security management. An illustrative example can be given with the case of connected devices, such as smartphones and smart objects, even if our work is not restricted to this specific context. The multiplication of these devices is an important factor of Internet growth. It also contributes to a larger attack surface, such as the recent attack against the Dyn DNS service, which relied on a botnet of infected smart objects. Malicious applications targeting these devices are also increasing massively every year. Preventive security methods that consist in analyzing the applications before their publication on markets show their limitations. For instance last year, Kaspersky Labs estimated that 2,961,727 malwares were published on the official Google application market [1], [2]. Security mechanisms must be dynamically adjusted to these evolutive threats. The resources of devices in terms of memory, cpu, and battery are also often limited, which may make the local deployment of protective mechanisms more challenging.

In the meantime, the programmability offered by software-defined networks (SDN) provides new perspectives with respect to protection [3], [4]. First, security mechanisms such as firewalls, intrusion detection systems, and data leakage prevention services, can be virtualized in cloud infrastructures, and be offered as outsourced services. We will refer to them as security functions in the remainder of this paper. Second, these functions can then be combined to build elaborate security chains that are deployed and adjusted, in a dynamic manner, to protect devices and their associated applications. These chains can be fully outsourced in the network, or with a hybrid in-cloud/on-device deployment scheme. Software-defined networking separates the control plane from the data plane and interconnects them by an open interface, which may typically be the OpenFlow protocol. In our case, the data plane corresponds to the forwarding switches with the security functions that are currently deployed in the network, whereas the control plane describes the orchestration underlying automatic modifications of the configuration of the data plane. In this way, different security functions can be combined as a chain and deployed in the network. This dynamic chaining allows for an adaptive response to attacks. However, it is crucial to ensure that these security chains, in view of their complexity and dynamics, are consistent. Misconfigurations may reduce their efficiency, and introduce security violations.

We therefore study in this paper the use of automated techniques for verifying security chains in software-defined networks. Our approach relies on a software-defined architecture and integrates formal verification methods for checking both the control and data planes. We introduce algorithms for translating specifications of security chains into formal models that are then automatically verified by SMT solving and model checking. Our solution has been prototyped as a package, named Synaptic, built as an extension of the Frenetic family of SDN programming languages. The solution is evaluated through extensive experiments using the checkers CVC4 [5], veriT [6] and nuXmv [7]. Our main contributions are: (i) the design of an automated verification strategy and its architecture, (ii) algorithms for translating specifications of security chains into formal models, (iii) the prototyping of the Synaptic package as an extension of the Frenetic language family, and (iv) the performance evaluation with different tools for SMT solving and model checking.

The remainder of this paper is organized as follows. Section II gives an overview of existing work in the area. Section III describes our automated verification strategy, its

architecture and the supporting algorithms for generating formal models based on security chains. It then presents our prototype, named Synaptic, built as an extension package for the Frenetic language family. Section IV describes performance results obtained with different verification tools. Section V concludes and points out future research perspectives.

II. Related work

The increasing development of cloud infrastructures has spurred the virtualization and outsourcing of a large variety of network services, including for security purposes. Network programmability provides a support for combining them in order to elaborate service chains. For instance, Sherry et al. [8] propose a solution for dynamically and transparently outsourcing middleboxes across several cloud providers using virtualization and different redirection mechanisms. Gibb et al. [9] present a similar approach where a cloud-based architecture is designed for outsourcing network functionalities using SDN. Regarding the chaining of such network functionalities, Qazi et al. introduces SIMPLE [10], a policy enforcement layer based on SDN and flow correlation for middlebox traffic steering. In the same vein, Fayazbakhsh et al. propose Flowtags [11], an architecture where middleboxes are extended to support OpenFlow and to deal with dynamic middlebox flow mangling. In the area of smart devices, Sapiol et al. [12] define a Network Function Virtualization (NFV) router to provide a per-user policy enforcement on mobile applications through service chaining. Previous work of our group [13], [14] proposed the design of security chains for protecting such devices, and showed their benefits. The dynamics and multiplication of elaborated chains require verification techniques to ensure their consistency.

There exists a substantial body of literature for supporting the verification of data planes in the area of software-defined networking. The purpose is typically to prevent misconfigurations that could lead to black holes and loops in the network. For instance, VeriCon [15] is a language for specifying software-defined networking policies and is accompanied by a solution for checking whether a policy verifies invariants expressed using predicate logic. It does not address the specification of temporal aspects of the control logic. NICE [16] is another example of SDN verification based on unit tests; however, tests cannot ensure complete coverage of the behavior of complex systems such as SDN policies. FlowChecker [17] represents the data plane as a binary decision diagram (BDD), whereas properties are expressed in computation tree logic (CTL). The complexity of the BDD-based modeling may constitute an obstacle to its large-scale deployment. VeriFlow [18] also proposes support for the real-time verification of OpenFlow rules, but its low level of abstraction may make it cumbersome to verify large security chains. Finally, VeriSDN [19] is another example of control plane verification in the specific case of SDN-based firewalls, although the verification of more sophisticated security functions is considered. In a similar spirit, work by Shaer et al. [20] considers anomaly detection in distributed firewalls. Most of these approaches are focusing on the verification of the data plane, and may miss dynamic aspects of the chain operating on the control and data planes.

Decoupling the functional specification of security chains and their translation into low-level configuration rules is an important requirement for enabling their formal verification, which motivates the use of high-level network programming languages. The Frenetic language family for network programming [21] counts among the most prominent approaches for verifying the control plane. It includes a language, called Pyretic, for specifying network configurations in Python; these configurations are then compiled into low-level rules [22]. Pyretic is complemented by an extension, called Kinetic, for describing control plane policies; Kinetic also offers formal verification techniques [23]. Whereas this work is restricted to the verification of the control plane, we argue in favor of verifying both the security functions chaining of the data plane and the dynamics of the control plane in an integrated manner in order to prevent errors due to potential misconfigurations.

III. Automated Verification of Security Chains

We propose in this paper an automated strategy for verifying both the control and data planes of security chains in software-defined networks before their deployment or re-configuration. Our goal is to analyze both the consistency of the data plane, taking into account its dynamic evolutions specified by the control plane. The solution is based on a dedicated architecture, which includes a verifier, named Synaptic, capable of translating security chain specifications into formal models that can be verified by SMT solving or model checking.

A. Proposed architecture

The architecture is depicted in Figure 1, with the illustrative context of protecting smart devices, such as smartphones. However, our work is not limited to this specific use case, since security chains are used more widely. The figure highlights three main entities: (i) on the left, the smart device, with several running applications, to be protected with an integrated agent (which can typically be instantiated by an OpenFlow virtual switch for redirection purposes); (ii) in the middle, a cloud provider infrastructure hosting several security functions as well as a security manager, and (iii) on the right, the remote destinations interacting with the applications executed on the smart device. The security manager is supported by an SDN controller that orchestrates security chains, using the Pyretic network programming language. In addition to the Kinetic extension that provides functions for verifying the control plane (blue), it exploits our Synaptic checker for verifying the data plane (shown in red). Finally security functions can be deployed either on the device or in a cloud infrastructure (dashed lines).

When an application initiates a communication with a remote destination, all the messages from and to that application go through the agent (virtual switch) of the device. The switch may probe the SDN controller from the cloud provider in order to know how to redirect the related messages for

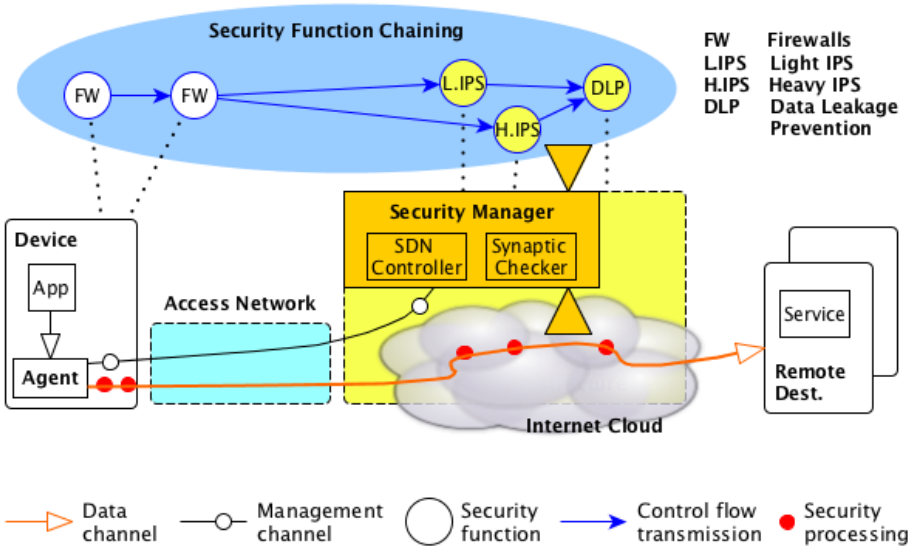

Figure 1. Our architecture supporting the orchestration and verification of security chains.

security checks. Depending on the risks and the context, the security manager activates the appropriate security functions. By pushing the necessary SDN rules within the cloud provider network, the controller composes the security functions to build an appropriate chain and notifies the switch. The latter directs the incoming and outgoing traffic through the chain before forwarding it to the final destination. Security functions can either be hosted locally on the smart device or in the cloud. Compositions of security functions are designed by the security manager according to several factors, including the originating application, the remote destination, and the network properties. For instance, the security functions may include network or applicative firewalls, intrusion detection systems, and mechanisms for preventing data leakage. They are not limited to traffic analysis, but may also include functions analyzing the configuration of smart devices and their applications. The design of these chains and the choice of security functions is out of the scope of this paper: we focus on their automated verification.

B. *Extension of the Frenetic framework*

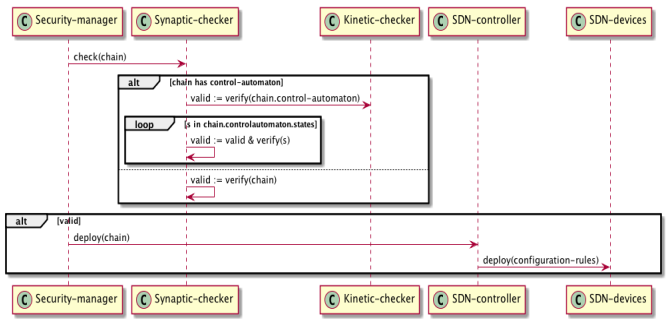

Figure 2. Interaction diagram among the components of our architecture.

Our solution extends the Frenetic framework. Specifically, the Pyretic network programming language is used for describing data plane configurations. The language includes four basic rules to express a static filtering on the traffic, namely: **identity** to forward all packets without conditions, **drop** to block all packets without condition, **match(x=y)** to forward only packets for which field x has the value y, and **modify(x=y)** to forward all packets and set the field x to the value y. They can be composed using three different composition operators: a sequential operator ($\gg$), a parallel operator ($+$) and a negation operator ($\sim$). In addition, the Kinetic language, also part of the Frenetic family, is intended for specifying and verifying the control plane, as a finite state machine. More specifically, it makes use of the NuSMV model checker to verify whether the corresponding automaton satisfies properties written as formulas of the temporal logic CTL. However, it does not verify the data plane.

We therefore define an extension of the Pyretic language capable of verifying the data plane. Our checker, that we call Synaptic, is based on a translation of the rules of this language into formal models. We consider two categories of formal models for supporting this verification. The first category corresponds to SMT solving. It is based on representing the conditions verified by the security chain as a set of logical constraints containing variables. This model is received as input by an SMT solver that checks its satisfiability by searching an assignment of the variables that makes all the constraints true. The second category is based on model checking. In that case, the rules are represented as a finite state machine, and the properties are specified in a (temporal) logic, similar to the verification of the control plane. Starting from this representation, a model checker builds the set of reachable states of the automaton, and checks the validity of properties, building a counter example when the property is violated.

Figure 2 illustrates the interactions among components in our architecture. The security chains that are specified in Pyretic are sent by the security manager to the Synaptic checker. They may first be checked using the Kinetic extension to identify inconsistencies in the control plane. The Synaptic checker then translates the chain specifications into a formal model, which is interpreted by an SMT solver or a model checker. The validation results are then sent back to the security manager, in order to determine whether the security chain can be deployed in the network. If this is the case, the security manager interacts with the SDN controller, which instantiates the chaining with the support of virtual switches.

C. Verification based on SMT solving

We first explain the verification of security chains based on SMT solving. In that context, the specification of a security chain is translated into the SMTlib input language of SMT solvers. Let x be a packet field, x_after be the same field after a modification, y be a value, and p_1 and p_2 be given policies. The translation function of Synaptic, in the following denoted by f, is defined in a quite straightforward way based on the grammar of Pyretic, in three main steps:

1) Translation of the Pyretic elementary rules into atomic propositions, as follows:
   - $f(identity) \rightarrow true$ and $f(drop) \rightarrow false$,
   - $f(match(x = y)) \rightarrow (= \ x \ y)$,
   - $f(modify(x = y)) \rightarrow (= \ x\_after \ y)$.
2) Translation of composition operators (sequence, parallel, and negation) into Boolean expressions, as follows:
   - $f(p_1 \gg p_2) \rightarrow f(p_1) \wedge f(p_2)$,
   - $f(p_1 + p_2) \rightarrow f(p_1) \vee f(p_2)$,[1]
   - $f(\sim p_1) \rightarrow \neg f(p_1)$.
3) Translation of properties of the security chain into SMTlib constraints. For a given security chain c and a property p, we want to make sure that the implication $c \rightarrow p$ is valid. This is equivalent to verifying that $c \wedge \neg p$ is unsatisfiable: the latter expression appears in the generated SMTlib input.

In order to illustrate this process, we consider a simple security chain composed of four security functions, noted F_i, with $i \in \{1, 2, 3, 4\}$ corresponding to only firewalls. However, our solution is not limited to a specific security function. The Pyretic specification corresponding to this chain appears in Listing 1. Each security function is described by composing the match rules that are applied to ip addresses and port values. For instance, the function F_3 only accepts packets whose source port is 4000, 5000 or 6000. These different functions are then combined using the sequence, parallel and negation operators to specify the final chain.

[1]The output of $p_1 + p_2$ is composed of the union of p_1 and p_2, which explains the translation of this operator as a disjunction.

```
F1 = match(srcip=IP("198.122.37.15")) +
     match(srcip=IP("253.182.3.14"))

F2 = match(srcport=1000) + match(srcport=2000) +
     match(srcport=3000)

F3 = match(srcport=4000) + match(srcport=5000) +
     match(srcport=6000)

F4 = match(dstport=7000) + match(dstport=8000) +
     match(dstport=9000)

chain = ((F1 ≫ F2)+(~F1 ≫ F3)) ≫ F4
```

Listing 1. Pyretic specification of a toy security chain.

The chain behavior can be described literally as follows: if the security function F_1 accepts a given packet, it is transmitted to the security function F_2; otherwise, it is transferred to the security function F_3. All packets that are accepted by F_2 or F_3 are finally transmitted to F_4. Assume that we want to check the following property of the chain: each packet accepted by F_1, F_2 and F_4 or rejected by F_1 and accepted by F_3 and F_4 must be accepted by the chain.

```
(set-option:produce-models true)
(set-logic QF_LIA)
; Declaration of variables and values
(declare-const allowed Bool)
(declare-const srcip Int)
(declare-const ip0 Int)
...
; Translation of the security chain
(assert (and
  (distinct port3 port5 port0 port7 ip1
            port8 port1 port4 port6 port2 ip0)
  (= allowed (and
     (or (and (or (= srcip ip0) (= srcip ip1))
              (or (= srcpt port0) (= srcpt port1)
                  (= srcpt  port2)))
         (and (not(or (= srcip ip0) (= srcip ip1)))
              (or (= srcpt port3) (= srcpt port4)
                  (= srcpt port5))))
     (or (= dstpt port6) (= dstpt port7)
         (= dstpt port8))))
; Translation of the property
 (and
   (or (and (or (= srcip ip0) (= srcip ip1))
            (or (= srcpt port0) (= srcpt port1)
                (= srcpt port2))
            (or (= dstpt port6) (= dstpt port7)
                (= dstpt port8)))
       (and (not(or (= srcip ip0) (= srcip ip1)))
            (or (= srcpt port3) (= srcpt port4)
                (= srcpt port5))
            (or (= dstpt port6) (= dstpt port7)
                (= dstpt port8))))
   (not allowed))))
(check-sat)(get-model)(exit)
```

Listing 2. SMTlib input generated for the toy example (excerpt).

Listing 2 shows the (slightly abridged) result of translating the toy security chain and the above property into the SMTlib language. It includes the declaration of variables and values (abstracting the concrete ip and port values), the translation of the security chain, and the translation of the (negated) property. In particular, each composition operator is translated

into a boolean operator, and the overall constraint represents the condition for a packet to be accepted by the chain. The generated file is fed to an SMT solver, which will return a verdict of unsatisfiability if the property is valid or else a model that encodes a counter-example to the property.

D. Verification based on model checking

Our Synaptic checker also supports the verification of security chains based on model checking. This is a direct complement to the existing functionality of Kinetic for verifying the control plane. However, the translation of a Pyretic specification into a finite state machine (or automaton), that can be verified by model checking, is less intuitive than the translation into an SMT model. In order to perform this translation, we extract strictly sequential sub-chains from the security chain described in Pyretic. The following inductive definition introduces the conditions for a chain C to be strictly sequential:

- $C \in \{identity, drop, match, modify\}$,
- $C = \sim C_1$ where C_1 is strictly sequential,
- $C = C_1 \gg C_2$ where C_1 and C_2 are strictly sequential.

Subchains built solely from basic rules, negation, and sequential composition are grouped together and define the transitions of the automaton, whereas the parallel composition forms the basis for defining the states of the automaton. Our translation algorithm therefore generates the data plane automaton from the Pyretic specification, based on the following steps:

1) Initially, generate the initial and final states of the data plane automaton.
2) Generate a variable corresponding to each packet attribute (e.g. srcip, srcport) that appears in the specification. These variables are used to specify transition conditions.
3) For each parallel composition, generate an automaton state. These states describe the position of a packet in the security chain. In particular, a packet is considered as accepted when it reaches the final state of the automaton.
4) For each strictly sequential sub-chain, generate an automaton transition. The conditions on this transition correspond to a combination of the elementary rules of the considered sequence.

Figure 3 illustrates the automaton obtained from the Pyretic specification for the example described in the previous subsection. The values for each packet attribute are abstracted by symbolic values, and the states associated to the security chain are noted $S_0, \ldots S_5$.

The resulting automaton is represented in a nuXmv model as a finite state machine. It is complemented by the specification of properties to be verified on the chain, written as temporal logic properties to be verified by the model checker. Listing 3 illustrates the properties obtained for our example.

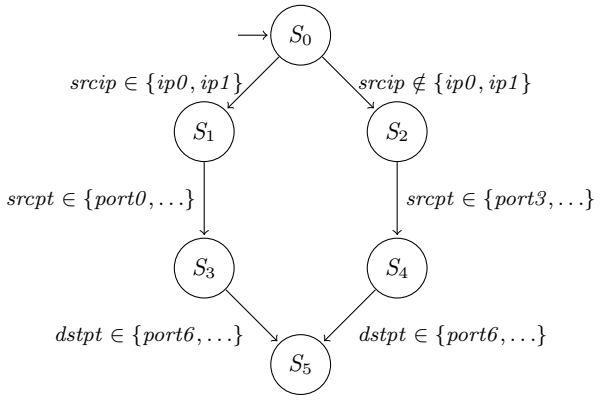

Figure 3. Data plane automaton for the toy example.

```
AG(((srcip=ip0 | srcip=ip1) &
    (srcpt=port0 | srcpt=port1 | srcpt=port2) &
    (dstpt=port6 | dstpt=port7 | dstpt=port8))
   -> EF state=S5)

AG(((!(srcip=ip0 | srcip=ip1)) &
    (srcpt=port3 | srcpt=port4 | srcpt=port5) &
    (dstpt=port6 | dstpt=port7 | dstpt=port8))
   -> EF state=S5)
```

Listing 3. Generated formal specification interpretable by a model checker.

These properties are expressed in the temporal logic CTL. The formulas are built up using the path quantifiers A and E and the temporal operators F and G. They assert that any data packet whose attributes satisfy the conditions of one of the subchains may reach the final state and therefore be accepted by the security chain. The state machine, together with the temporal formulas, can be passed to nuXmv in order to verify the same properties as the ones verified by SMT solving in the previous sections. Alternatively, properties could be expressed in LTL (linear-time temporal logic), or they could express safety properties that the automaton should satisfy in every state.

E. Implementation of Synaptic

We have implemented our Synaptic checker and its translation algorithms, as a verification package written in Python. It has been designed as an extension of the Pyretic SDN programming language, and exploits the Kinetic extension, part of the Frenetic language family. Our package can be used to verify both the control and data planes of security chains built using the Pyretic language. As depicted in Figure 2, it takes as input the specification of a given security chain expressed in Pyretic, and provides back to the security manager the verification results, corresponding to a verdict on the validity of the chain. Based on this analysis, the security manager can then interact with the SDN controller, so that the security chain is deployed in the programmable network.

Internally, the Synaptic checker relies on two main building blocks: a semantic analyzer capable of interpreting the security chain and its symbols, and a model generator capable of

generating a given formal model. The model generator is defined in an abstract manner, so that our checker can easily be extended to other formal languages. We have currently implemented two model generators corresponding to each of the translation algorithms presented below:

- an SMTlib model generator, which produces a formal specification interpretable by an SMT solver, including CVC4 and veriT,
- a nuXmv model generator, which produces a formal specification interpretable by the nuXmv solver.[2]

Our checker can first verify the control plane associated to the security chain, with the support of the Kinetic extension. It then generates the formal specifications corresponding to the data plane, and transmits them to the corresponding solver. The verification of a given security chain can be performed pro-actively by exploring all the possible states reachable by the control automaton, and then all the possible data planes, or can also be performed more reactively by only considering a subset of possible data planes depending on the changes operated on the security chain.

IV. Performance Evaluation

We evaluated the performance of our prototype through an extensive series of experiments. In particular, we wanted to compare the performances obtained with the two translations to SMTlib and nuXmv, as well as for different SMT solvers in order to evaluate the overhead introduced by Synaptic. The experimental setup was based on a MacBook Air laptop computer with a Intel Core i5 (1.7 GHz) processor, and 4Gb of RAM memory. We considered the three following solvers: CVC4 (version 1.4), veriT (version 201506), and nuXmv (version 1.0.1). In order to perform these experiments, we have implemented an additional Python module capable of generating synthetically the security chains that are used as inputs of our Synaptic checker. This generation takes into account several parameters, including:

- the size of the control plane automaton specifying the changes to the security chain in response to network events, measured as the number of states;
- the size of the security chain expressed in terms of both width and length;
- the number of properties that have to be verified by the checker on the security chain.

While varying these different parameters, we evaluated the response time and memory consumption observed with SMT solving and model checking. This evaluation includes the translation of the specification into a given formal model, and its checking by a given solver. We used the Python time module and the valgrind memory profiler to perform our measurements and obtain statistically grounded results.

[2]We use nuXmv rather than its predecessor NuSMV used by Kinetic in order to directly express the constraints on packet attributes.

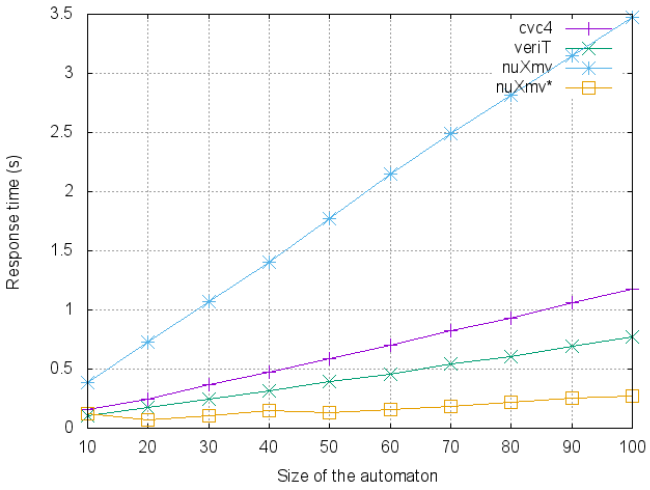

Figure 4. Response time vs. size of the control automaton.

A. Impact of the control automaton size

In a first series of experiments, we are interested in quantifying the impact of the size of the control automaton on the verification performance. Our checker supports both the verification of the control and data planes. The size of the automaton directly influences the number of processes required for verifying the security chain. Consider the case of a security chain with an automaton composed of n states. The Synaptic checker therefore must perform $n+1$ verification processes: one supported by the Kinetic extension to check the control plane, and n processes corresponding to the different automaton states to check the data plane. We use our Python module to generate security chains with different control automaton sizes, varying from 0 to 100 states, in order to quantify the response time and memory consumption induced by our Synaptic checker.

Figure 4 represents the response time (in seconds) of our checker with different backend solvers: CVC4, veriT, and nuXmv. As the verification of the control plane is also based on model checking, it is possible to execute the verification of both control and data planes in a single nuXmv instance. This case corresponds to the last curve, noted nuXmv*. According to these results, we can observe that the response time grows linearly with the size of the control automaton for each of the verification methods. The best performance with that respect is obtained with the nuXmv* configuration, where the control and data planes are verified in the same nuXmv instance. Surprisingly, the worst case is given by the nuXmv curve, with one verification process per automaton state. In that case, the response time is nevertheless still quite acceptable, with a total time of 3.5 seconds for a control automaton of 100 states.

We also quantify the impact on the memory consumption with these different automaton sizes. Figure 5 indicates the maximal memory consumption required during the verification process. Only two curves are visible in this figure, one corresponding to the nuXmv* verification, and another one

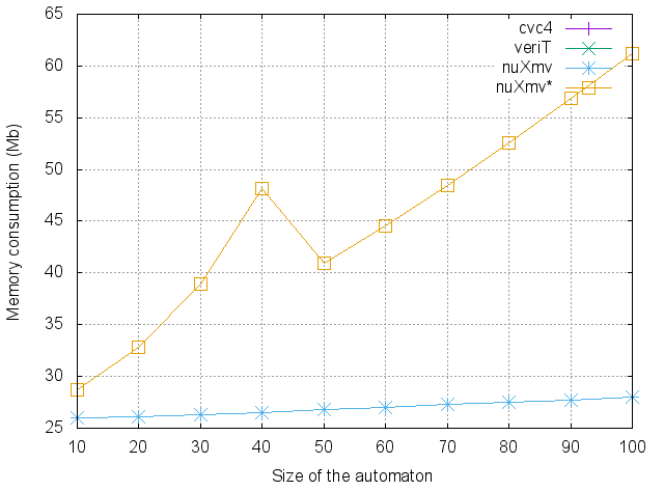

Figure 5. Memory consumption vs. size of the control automaton.

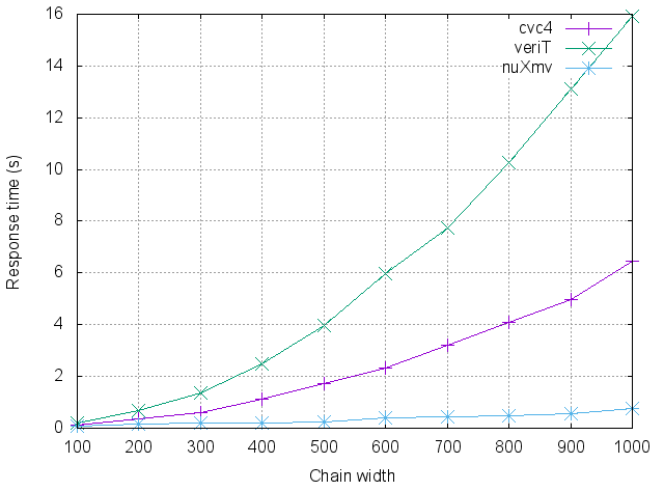

Figure 6. Response time vs. width of the security chain.

corresponding to the performance of CVC4, veriT and nuXmv, which are identical. For these three last approaches, the highest memory consumption is always generated by the process supporting the verification of the control plane, using the Kinetic extension and the nuXmv model checker. In these experiments, the nuXmv* approach consumes more memory, with a supra-linear behavior when varying the size of the control automaton. In particular, we obtain a memory consumption of 62 Mb for a 100-state automaton. The three other curves (CVC4, veriT, nuXmv) are characterized by a linear behavior, with a memory consumption of about a third of that of nuXmv* for the same 100-state automaton. Therefore, the low response time given by nuXmv* is balanced by a higher memory footprint, which can constitute a limiting factor in scenarios where many security chains have to be checked: for instance, in the case of several security chains protecting several individual applications on a set of smart devices.

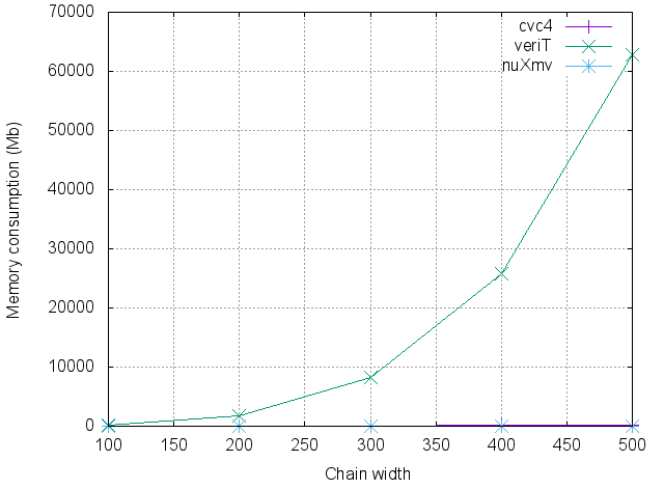

Figure 7. Memory consumption vs. width of the security chain.

B. Impact of the width and length of the security chain

In a second series of experiments, we evaluate to what extent the width and length of the security chain impact the performance of our Synaptic checker. As we are only focusing on the data plane in these experiments, we do not distinguish the nuXmv and nuXmv* cases. We therefore only consider the verification results obtained with the backends CVC4, veriT, and nuXmv.

We first study the impact of the width of the security chain, corresponding to the number of functions (or rules) to be composed in parallel. Figure 6 illustrates the response time of our prototype, while varying the width of the security chain from 100 to 1000. The observed response times grow slightly more than linearly for the three verification methods. We expected the same phenomenon as for the experiments done with the control automaton. In fact, the best performances are obtained with nuXmv. This latter provides a value of 0.8 seconds for a security chain width of 1000, while CVC4 and veriT generate values of respectively 6.4 and 15.9 seconds for the same conditions. When we analyze the results with respect to memory consumption, which are shown in Figure 7, we can observe that veriT requires 62.81 Gb of memory for the widest security chains. CVC4 and nuXmv are much more efficient and only consume respectively 29.89 Mb and 202.67 Mb in the worst cases.

We then focus on the length of the security chain corresponding to the number of functions (or rules) that are composed sequentially in the chain. Figures 8 and 9 illustrate respectively the performance in terms of response time and memory consumption, considering a security chain with a length varying from 10 to 100 functions. This parameter has an important impact on the response time of CVC4, with a value of more than 100 seconds in the worst case, while the two other approaches, nuXmv and veriT, maintain acceptable values in these conditions. The results on memory consumption are only indicated for the backends nuXmv and veriT, with CVC4 being disqualified by its bad timing behavior. It

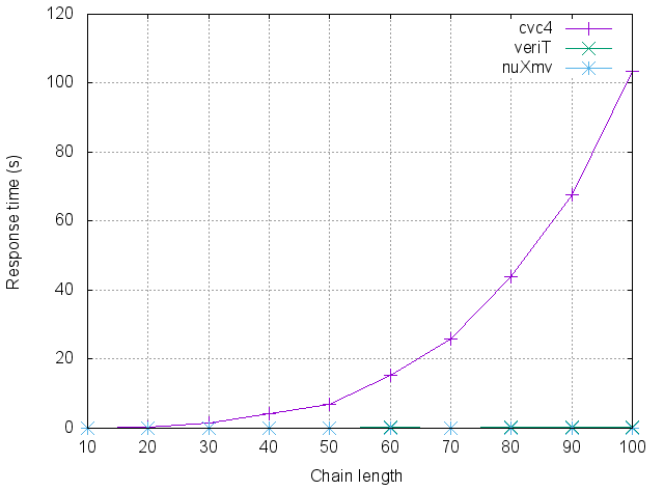

Figure 8. Response time vs. length of the security chain.

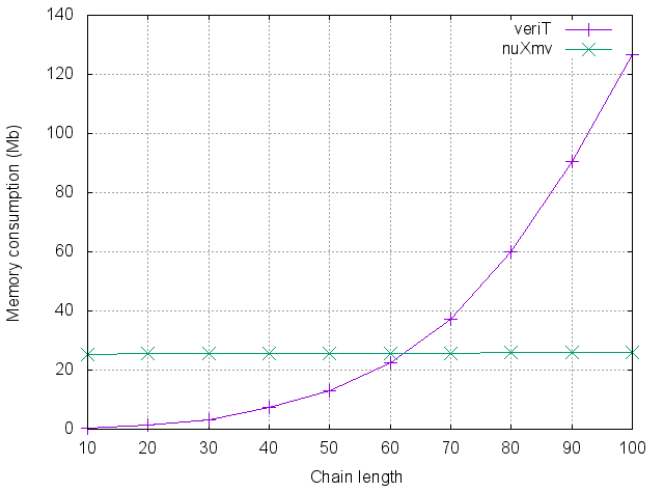

Figure 9. Memory consumption vs. length of the security chain.

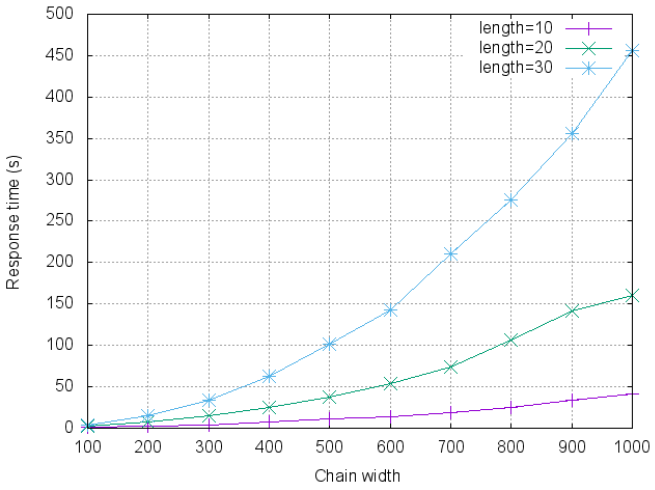

Figure 10. Response times of nuXmv vs. both width and length.

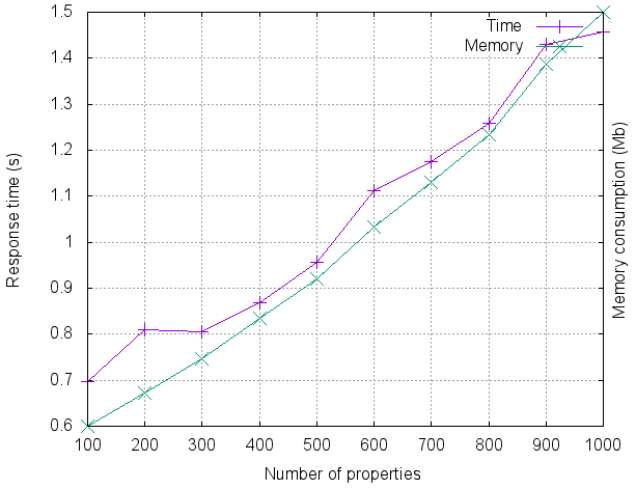

Figure 11. Performance of nuXmv vs. number of properties to be checked.

appears that veriT consumes less memory than nuXmv for the smallest security chains, from a size of 10 to 60. However, when the length increases from around 80 to 100, nuXmv shows the best results, with a stable value around 26 Mb, while veriT requires almost 127 Mb in the worst conditions. These series of experiments on length and width go in favor of nuXmv (model checking), rather than CVC4 and veriT (SMT solving), when addressing complex security chains.

To complement these results, we perform a third series of experiments, where we quantify simultaneously the impact of the width and length of the security chains for the nuXmv backend, which provided the best performances so far. Figure 10 gives the observed response times with a width varying from 100 to 1000, and a length varying from 10 to 30 (corresponding to the different plotted curves), representing a total number of rules from 1000 to 30000. The most complex security chains require more than 450 seconds of verification time. Some additional optimizations with respect to nuXmv formal modelling could be envisioned to reduce these values, such as reducing the domains associated to packet field representations, or reducing the number of initial states based on the requirements induced by the properties that have to be checked.

C. Impact of the number of properties to be checked

In a last series of experiments, we study more specifically the impact of the number of properties that have to be satisfied by the security chain. Again, due to the previous performance results we only consider the nuXmv backend, where properties are expressed in the temporal logic CTL.

Figure 11 shows both the response time and memory consumption generated by our Synaptic checker with the nuXmv backend, considering a security chain composed of 1000 functions, and varying the number of properties to be checked from 100 to 1000 properties. The performances appear to be linear with respect to the number of properties. More precisely,

the response time varies from 0.69 to 1.46 seconds, while the memory consumption goes from 43.34 to 173.99 Mb.

These performance results could be improved depending on the nature of the properties. For instance, when checking simple invariants rather than more complex temporal properties, we could use more efficient dedicated algorithms available in nuXmv.

V. Conclusions

We have proposed in this paper an automated approach for verifying security chains that are deployed in software-defined networks. Our solution takes into account both the control and data planes of these critical chains, which combine different security functions, such as firewalls, intrusion detection systems, and services for preventing data leakage. We have described the architecture supporting our technique, as well as the behavior and interactions among its different components. In particular, we have presented our Synaptic checker, which is capable of generating formal models from the security chain specifications, through an extension of the Frenetic language family. We have designed and implemented translation algorithms that underlie verifications of the data plane based on SMT solving and model checking, complementary to the verification of the control plane performed using Kinetic. We developed a prototype of our solution implementing these algorithms, and evaluated its performances through extensive series of experiments based on the backend solvers CVC4, veriT, and nuXmv. The experiments showed the benefits and limits of these methods in terms of response time and memory consumption while varying different sizes of security chains, and numbers of properties to be checked on them.

As future work, we are working on an extension of our translation algorithms to support more complex and advanced rules associated to the security functions. We are also interested in pursuing the optimizations of formal models that we are generating automatically, based on the nature of checked properties. Finally, we want to investigate further the integration of this automated verification process into the context of an automated management framework for security chains.

References

[1] M. La Polla, F. Martinelli, and D. Sgandurra, "A Survey on Security for Mobile Devices," in *IEEE Communications Surveys & Tutorials*, 2012.

[2] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android Security, a Survey of Issues, Malware Penetrations and Defenses," in *IEEE Communications Surveys & Tutorials*, July 2015.

[3] 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.

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

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proc. 23rd Intl. Conf. Computer Aided Verification (CAV 2011)*, Snowbird, UT, USA, 2011, pp. 171–177.

[6] T. Bouton, D. C. B. D. Oliveira, D. Déharbe, and P. Fontaine, "veriT: An Open, Trustable and Efficient SMT-Solver," in *Proc. 22nd International Conference on Automated Deduction (CADE-22)*, Montreal, Canada, 2009, pp. 151–156.

[7] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Proc. 26th Intl. Conf. Computer Aided Verification (CAV 2014)*, Vienna, Austria, 2014, pp. 334–342.

[8] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," in *Proc. ACM SIGCOMM Internationl Conference (SIGCOMM'12)*, Helsinki, Finland, 2012, pp. 13–24.

[9] G. Gibb, H. Zeng, and N. McKeown, "Outsourcing Network Functionality," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks (HotSDN'12)*. ACM, 2012, pp. 73–78.

[10] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement using SDN," in *Proc. ACM SIGCOMM International Conference (SIGCOMM'13)*, Hong Kong, China, 2013, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486022

[11] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags," in *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, Seattle, WA, USA, 2014, pp. 543–546.

[12] A. Sapio, Y. Liao, M. Baldi, G. Ranjan, F. Risso, A. Tongaonkar, R. Torres, and A. Nucci, "Per-user Policy Enforcement on Mobile Apps Through Network Functions Virtualization," in *Proc. 9th ACM Workshop on Mobility in the Evolving Internet Architecture (MobiArch'14)*, Maui, HI, USA, 2014, pp. 37–42.

[13] 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.

[14] ——, "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.

[15] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "Vericon: Towards Verifying Controller Programs in Software-Defined Networks," in *Proc. 35th ACM SIGPLAN Intl. Conf. Programming Language Design (PLDI'14)*, Edinburgh, UK, 2014, pp. 282–293.

[16] M. Canini, D. Venzano, P. Pereskini, D. Kostic, and R. Jennifer, "A Nice Way to Test OpenFlow Applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, 2012.

[17] E. Al-Shaer and S. Al-Haj, "FlowChecker, Configuration Analysis and Verification of Federated OpenFlow Infrastructures," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration (CCS'10)*, 2010.

[18] A. Khurshid, X. Zou, W. Zhou, M. 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.

[19] M.-Y. Kang, J.-Y. Choi, I. Kang, H. H. Kwak, S. J. Ahn, and M.-K. Shin, *A Verification Method of SDN Firewall Applications*. IEICE Transactions on Communications, 2016.

[20] E. S. Al-Shaer and H. 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 2004)*, 2004.

[21] 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.

[22] 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.

[23] 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.