Improving the formal verification of reachability policies in virtualized networks

This article has been accepted for publication in a future issue of this journal, but has not been fully edited.
Citation information: DOI 10.1109/TNSM.2020.3045781, IEEE Transactions on Network and Service Management

Improving the formal verification of reachability policies in virtualized networks

Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Serena Spinoso, Fulvio Valenza, Jalolliddin Yusupov

Abstract—**Network Function Virtualization (NFV) and Software Defined Networking (SDN) are new emerging paradigms that changed the rules of networking, shifting the focus on dynamicity and programmability. In this new scenario, a very important and challenging task is to detect anomalies in the data plane, especially with the aid of suitable automated software tools. In particular, this operation must be performed within quite strict times, due to the high dynamism introduced by virtualization. In this paper, we propose a new network modeling approach that enhances the performance of formal verification of reachability policies, checked by solving a Satisfiability Modulo Theories (SMT) problem. This performance improvement is motivated by the definition of function models that do not work on single packets, but on packet classes. Nonetheless, the modeling approach is comprehensive not only of stateless functions, but also stateful functions such as NATs and firewalls. The implementation of the proposed approach achieves high scalability in complex networked systems consisting of several heterogeneous functions.**

Index Terms—**network reachability, data plane verification, service function chains, network security policies**

I. Introduction

Nowadays, *Network Functions Virtualization* **(NFV) [1] and** *Software-Defined Networking* **(SDN) [2] are heavily changing computer networks. These paradigms raised flexibility and agility in service function deployment, since each function is a software program running on a general-purpose server. Besides, high programmability is provided for traffic steering, since traffic can be dynamically redirected in different ways (e.g., on a per-user and per-flow basis).**

Modern virtualized networks are not based only on switches and routers, but they also include a growing number of complex service functions [3] (e.g., firewalls, DPIs, NATs) that must be properly configured to perform their own processing and forwarding operations on incoming packets. However, the NFV/SDN paradigm also introduces new issues and challenges in the management of this configuration process. In particular, network function configurations may be updated very frequently in such a highly flexible environment. Consequently, preventing erroneous forwarding behaviors of the network, such as unreachable destinations or non-perfect isolation of sub-networks [4], becomes a complex task.

D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, and F. Valenza are with the Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy (e-mail: daniele.bringhenti@polito.it, guido.marchetto@polito.it, riccardo.sisto@polito.it, serena.spinoso@polito.it, fulvio.valenza@polito.it).

J. Yusupov is with the Department of Automatic Control and Computer Engineering, Turin Polytechnic University in Tashkent, Tashkent, Uzbekistan (e-mail: jaloliddin.yusupov@polito.uz).

Currently, the low-level configuration of many *Virtual Network Functions* **(VNFs) relies on specific parameters, whose values have to be selected and set manually by the network administrator. In practice, this implies a typical configuration approach by trials and errors. Frequently, in the context of telco operators, when a misconfiguration is detected, administrators try to correct errors by manually changing the configurations of some functions and repeat this process until no more anomalies are observed. This technique, besides being cumbersome and time-consuming, lacks a comprehensive view of the network behavior and, as such, is error-prone and can make the network significantly hard to maintain.**

Because of this, suitable automated software tools are required, enabling operators to check networks before service deployment, thus preventing the violation of security properties and service downtimes. Many solutions are available in this field (e.g., [5], [6], [7], [8]), all based on ad-hoc formal descriptions and sound theoretical foundations, i.e., on a formal verification approach. Although formal verification is very powerful to satisfy the above requirements, there are other challenges that are often difficult to meet. NFV and SDN have introduced higher flexibility and dynamicity for network management [1]. This native characteristic of virtual networks is coupled with the improvement of intrusion detection and reaction techniques, such that, every time an intruder is detected, the structure and configuration of the network is quickly modified [9]. In this context, the network topology might undergo many changes in succession. As a consequence, network verification solutions must work within quite strict times, to support the ever-changing nature of virtualized networks. Moreover, they must be able to deal with the full variety of middleboxes typically used in such networks. Virtualization has, in fact, eased the creation of more complex network functions [10], and modeling their behavior for the application of formal verification techniques has become a hard task.

In this paper, we propose a formal verification approach relying on network models that enables the formal verification of reachability properties (named policies) against a network composed of both stateless and stateful network functions. This modeling approach, in addition to having wide coverage of network function types, is highly performant with respect to other strategies ([6], [7], [8], [11], [12]) that have been proposed in literature. A first reason is that the policies are verified for packet classes, which can represent multiple packets at the same time. A second motivation is that, under specific conditions, the verification of each policy is performed chain-by-chain, analyzing only the chains that are relevant for

one side allows a larger pool of network functions models, and on the other side it is characterized by richer formal models for the VNFs, so that these models are closer to the effective behavior of the corresponding network service functions. Finally, even though these models are richer and more accurate, the scalability which is achieved by our methodology is better, in terms of performance, thanks to the modeling of traffic flows rather than single packets and the avoidance of recurring to LTL, which is used in [8] but not in our approach.

III. Approach

Verifying in a formal and provably correct way that the packets originated by the source of an end-to-to-end communication can reach their destination is a complex task. Network virtualization worsened this issue, all the more. On one side, the variety and heterogeneity of the network functions are heightened, given that creating different software programs is easier with respect to older hardware implementations. On the other side, the configuration of each function is more complex, due to the growing size of modern virtualized networks, and can be dynamically changed, as enabled by their intrinsic reactivity.

In view of this problem, we propose an optimized approach to formally and rapidly verify the reachability between network nodes or subnetworks through stateless and stateful configured virtual functions. This approach lays its foundations on the definition of a decision problem called *Satisfiability Modulo Theories* (SMT). Additionally, the verification is based on policy-based management: the communications subject to verification are identified by reachability policies[2]. Each policy requires that the packets whose characteristics are denoted by the policy itself can (or cannot) reach their final destination. Each policy can altogether identify multiple packet classes at the same time (e.g., the reachability for all the packets coming from the subnetwork 150.23.1.0/24 to the server 20.22.2.2 can be expressed with a single policy).

High performance is reached mainly thanks to two features characterizing our approach. The first is that the reachability verification can be applied for packet classes, rather than single packets. The second is that we exploit the fact that, under specific conditions that will be detailed later, the verification can be performed chain-by-chain rather than considering the whole graph. This entails that, after identifying all the possible chains (i.e., the paths in-between the source-destination pair) which a communication may cross in the network, reachability is checked on such relevant chains only, and in parallel with one job per chain, rather than on the whole subgraph containing them. As it results from the formal proof given in [8], in most of the real-world circumstances the verification chain-by-chain provides the same results as the verification on the whole network.

In the remainder of this section, we describe how the verification workflow is organized in Section III-A, and how verification is formally achieved for each policy by solving a decision problem in Section III-B. Then, we use Section IV, Section V, and Section VI to exhaustively illustrate the formal models, the definitions and the formulas on which the SMT problem is based.

[2]In the rest of the paper, the term policies stands for reachability policies.

A. Verification workflow

The verification process proposed in this paper requires a set of reachability policies, a virtual network graph (i.e, the logical representation of the virtual functions and their interconnection), and a description of the configuration for each VNF as inputs. Then, the process establishes which policies are satisfied (or not), alongside with the related reasons. Note that verification is applied on the virtual network graph, before the deployment of the VNFs. As widely recognized now, an earlier verification is much better for network administrators to speed up error detection.

For each policy, the verification process is structured into two sub-tasks: chain *extraction* and *verification*. The first task, chain extraction, simply consists in identifying all the possible paths for the source-destination pair identified by the policy. Note that each element of this pair can be a single network node or, alternatively, a subnetwork. Chain extraction must be performed any time the network topology changes, because that event may have an impact on the interconnection between source and destination. Even if this task introduces an overhead in the verification process, its presence simplifies and optimizes the performance of the whole process. This is because changes in a network graph are less frequent than changes in service function configurations; then, chain extractions are less frequent than chain verifications. Moreover, using this approach, each extracted service function chain is loop-free by construction.

The second task, i.e. reachability verification, is then performed for each identified chain in parallel, or on groups of chains in the rare cases when chain-by-chain verification is not possible. For each chain, the traffic flows characterized by the packet classes identified by the policy are determined, and then, it is checked if they can reach the destination of the chain by solving an SMT problem. This parallelism enables an increase of performance, which sums up to the improvement that is achieved by checking the reachability of packet classes instead of each specific packet.

Nevertheless, there are conditions to be satisfied so that a verification performed chain-by-chain, such as the approach proposed in this paper, provides a result that applies to the whole network graph. More specifically, the optimization that consists of verifying chains separately and in parallel can be applied only when all the middleboxes in the chains take decisions on the received packets only depending on the state related to the flow to which the packets belong. In other words, each middlebox could be replaced by multiple middleboxes of the same type, each one dealing with a different flow in parallel, without impacts on the outcome of the forwarding decisions. In case the middleboxes that are present in the chains where a policy must be verified do not satisfy these conditions, instead of being verified chain-by-chain, the reachability property should be verified on the whole subgraph (i.e., a directed acyclic graph) that includes all the chains that pass through those middleboxes.

These conditions for performing a chain-by-chain verification are the same as those analyzed in [8], where a more formal specification of the conditions is provided and their validity is formally proved. As it results from their proof, when the conditions are met, if a reachability policy is valid for each chain, it is valid also for the whole graph. The same paper also shows that the conditions are satisfied by most real networks, thus making this simplification possible in most real cases.

B. Verification through SMT problem

The Boolean Satisfiability (SAT) problem is a traditional problem in mathematical logic and computer science. The aim of a SAT problem is to identify if, given a set of Boolean formulas, there exists an interpretation (i.e., an assignment of the variables appearing in the formulas), that determines the truth of all the formulas. The formulas of a SAT problem are expressed in propositional logic; therefore, they are composed by Boolean variables and the three classic logic operators ($\wedge$, $\vee$, $\neg$). Instead, SMT is a decision problem that applies to First Order Logic (FOL) theories, i.e. logic systems where formulas contain, in addition to the boolean operators, also certain constants, variables, functions and predicates defined over domains such as integers, strings or other data structures, and quantifier operators ($\exists$, $\forall$).

The main advantage of formulating the verification problem as SMT instead of SAT is thus a greater expressive power, which is fundamental to define models that support the complexity and heterogeneity of the network functions. Regarding possible performance issues, even though the SMT problem belongs to the NP-complete computational class, its worst-case complexity is not indicative of its average complexity, so that state-of-the-art SMT solvers usually show good performance and scalability in solving problems [22], thus making them a valid alternative to other approaches such as model checking based on state exploration. In fact, the latter requires time and memory that usually increase exponentially with system complexity, while the SMT-based approach is less prone to this problem.

Given a set of FOL formulas, the SMT problem consists of finding if there exists an interpretation of free constants, functions and predicates that makes all the formulas true. In this case, the SMT solver returns SAT, and provides an interpretation that makes all formulas true, otherwise it returns UNSAT.

For the SMT problem defined to verify reachability on a chain or group of chains, the following elements are modeled as sets of FOL formulas: the policy, the packet classes the policy refers to, the chain itself, with its forwarding principles, and the forwarding behavior and configuration of each network function in the chain. In these formulas, some predicates and constraints over the predicates are defined. Then, an off-the-shelf SMT solver is used to search for an interpretation that satisfies all the constraints. The definitions and formulas of this SMT problem are presented in the reminder of the paper. For the sake of clearness, they are divided into three groups, each one corresponding to a specific set of model elements. In greater detail, the definitions and formulas related to the basic network elements (i.e., network topology and traffic flows) are described in Section IV, those regarding the specific behavior of each network function type in Section V, and those concerning the reachability properties that must be verified in Section VI.

IV. Network Modeling

In this section, we present the models of the network topology, the network chains, and the traffic flows.

A. Network model

Formally, a directed graph G is a pair of sets $(\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the set of vertices and $\mathbb{E}$ is the set of edges, formed by pairs of vertices. In this paper, the network is modeled as a graph G, where the vertices represent network nodes that can send and receive packets, whereas the directed edges represent connections between network nodes, i.e. an edge connecting node n_1 to node n_2 means packets can flow from n_1 to n_2. Note that in this model we do not represent explicitly the interfaces of a network node, but we simply represent its ingress and egress connections.

Each node of the graph is characterized by a single IP address, by an IP address range or, more generically, by a set of IP addresses. Besides, the nodes are classified into two main types: end-hosts and service functions (i.e., $\mathbb{V} = \mathbb{H} \cup \mathbb{M}$). In brief, an end-host $h \in \mathbb{H}$ is a network node, a gateway or a subnetwork where communications originate or terminate. Instead, a service function $m \in \mathbb{M}$ is any intermediary middlebox placed in between end-hosts, which performs functions other than the normal, standard functions of an IP router on the datagram path between a source and destination host.

A network chain $c \in \mathbb{C}$ is a path in G, where the first and last elements are end-hosts, while the others are service functions. In each network chain, no element appears more than once. Formally, $\mathbb{C}$ denotes a set of chains, and c_i a single chain, which is modeled as a tuple made of the source end-host, the destination end-host and the ordered sequence of intermediate functions that make the chain. In formulas:

$$\mathbb{C} = \{c_1, c_2, ...\} \quad c_i = (s_i, d_i, M_i) \quad M_i = [m_{i1}, m_{i2}, ...]$$
$$c_i \in \mathbb{C}, \quad s_i \in \mathbb{H}, \quad d_i \in \mathbb{H}, \quad m_{ij} \in \mathbb{M}$$

In the remainder of this paper, the "." notation, when applied to a tuple, is used to retrieve a specific element. For example, $c_i.s_i$ represents the source end-host of the chain c_i.

B. Traffic flow model

We define an atomic traffic flow $f \in \mathbb{F}$ an end-to-end flow of packets directed from a source $v_s \in \mathbb{H}$ to a destination $v_d \in \mathbb{H}$, which is steered along the same chain $c \in \mathbb{C}$. The formal model of an atomic flow $f \in \mathbb{F}$ is a list

$$f = [v_s, t_{sa}, v_a, t_{ab}, v_b, ..., v_k, t_{kd}, v_d]$$

where each $v_i \in \mathbb{V}$ is a node on the chain crossed by the flow, while each t_{ij} represents a class of packets (called traffic in this paper) transmitted from v_i to v_j. In particular, each t_{ij} is the result of an action that node v_i applies to the traffic t_{ki} that precedes it in the list. The traffic received by each node

of a flow can thus be different from the other ones because of active service functions that can modify some packet fields. Besides, the packets of the ingress traffic of each node of an atomic flow are all managed in the same way by the node, i.e. the decision taken by the node on each one of them is the same. For conciseness, in the reminder of the paper the atomic traffic flows will be simply referred to as flows.

Let then $\mathbb{T}$ be the set of all the packet classes in the network. In order to model packet classes, a pre-defined set of packet fields or packet attributes, denoted as $\mathbb{A}$, is considered. Here, for simplicity, we consider only the following ones, but the methodology is fairly independent from what fields or attributes are selected:

- IPSrc and IPDst represent the source and destination IP addresses of the packet class;
- pSrc and pDst represent the source and destination transport-level ports of the packet class;
- tProto represents the transport-level protocol of the packet class;
- domain represents the domain related to the web communication which the packet class may represent;
- url represents the web location of the resource with respect to which the communication represented by the packet class happens;
- appInfo represents other header information related to the application-level protocols of the packet class;
- body represents the IP payload.

Each packet class $t \in \mathbb{T}$ is modeled as a logical conjunction of predicates, each one depending on variables representing the values of specific packet fields or attributes. Let us denote P_t the set of predicates that model the traffic $t \in \mathbb{T}$, X_t the set of variables on which at least a predicate in P_t is defined over. The information about which packet field or attribute a predicate represents can be retrieved with the $\alpha : P_t \rightarrow \mathbb{A}$ function. Instead, the variables on which a predicate is defined can be retrieved with the $\xi : P_t \rightarrow \mathcal{P}(X_t)$ function. Each predicate can be, in fact, defined over multiple variables: for example, for a predicate representing an IPv4 address, four variables are used, each one of them for a byte of the address. Besides, note that not all the packet fields and attributes that have been listed before must be modeled as predicates in the P_t set of each packet class, because some of them may not be useful for a specific traffic identification. For instance, predicates for url and domain fields are only defined for packet classes representing web communications.

In the following, some examples of predicates are illustrated.

- In a predicate with the form $x = value$, the packet field or attribute is bounded to have the specified value. Examples are "IPSrc = 150.10.0.1" and "domain = google.it". The first example is formulated with a sugared syntax, an abuse of notation that is used in the remainder of the paper to make the model presentation easier to understand. In particular,"IPSrc = 150.10.0.1" stands for the predicate $x_1 = 150 \wedge x_2 = 10 \wedge x_3 = 0 \wedge x_4 = 1$, where x_i is the variable representing the i-th byte of the source IP address packet field.
- In a predicate with the form $x \in set$, the packet field or attribute is bounded to have a value that belongs to the specified set. When the $*$ symbol is used as set, it means that the variable can be assigned any legit value depending on the predicate's type (e.g., for tProto, $*$ stands for the set of all the possible transport-layer protocols). Examples are "IPSrc $\in$ {150.10.0.1, 150.10.0.7}" and "domain $\in$ {google.it, youtube.com}".
- In a predicate with the form $x \subseteq range$, the packet field or attribute is bounded to have a value that falls into the specified range. This kind of predicate can be applied only to interval variables, whose value can be a range. When the $*$ symbol is used as range, it means that the variable can be assigned any subset of values depending on the predicate's type (e.g., for pSrc and pDst, $*$ stands for the full range [0, 65535]). Examples of predicates could be "IPSrc $\subseteq$ 150.10.0.0/24" and "pSrc $\subseteq$ [80-110]". The formulation of the predicate regarding the range of values the source tranport-level protocol can have is another example of the sugared syntax used in this paper. In this case, "pSrc $\subseteq$ [80-110]" stands for the predicate $x \geq 80 \wedge x \leq 110$, where x is the variable representing the port.
- For each packet field or attribute, specific predicates may be defined. For example, considering the string type x variable representing the IP payload, the contains(x,s) predicate is true when the x variable contains the s value.

In addition, some functions are introduced to complete the description of the traffic model. Their role is to enable retrieving useful information from each traffic flow f:

- π: $\mathbb{F} \rightarrow (\mathbb{V})^*$ retrieves the ordered list of nodes belonging to the $\mathbb{V}$ set which are crossed by a specific flow $f \in \mathbb{F}$.

$$\pi([v_s, t_{sa}, v_a, t_{ab}, v_b, ..., v_k, t_{kd}, v_d]) = [v_s, v_a, v_b, ..., v_k, v_d] \quad (1)$$

- τ: $\mathbb{F} \times \mathbb{V} \rightarrow \mathbb{T}$ retrieves the traffic $t \in \mathbb{T}$, belonging to a specific flow $f \in \mathbb{F}$ and received by a specific node $v \in \mathbb{V}$. If the flow does not pass through v, then this function returns t_0, a special element of $\mathbb{T}$ representing the packet class containing no packet (i.e., absence of traffic).

$$\tau([v_s, t_{sa}, v_a, t_{ab}, v_b, ..., v_k, t_{kd}, v_d], v_b) = t_{ab} \quad (2)$$

- ϕ: $\mathbb{V} \times \mathbb{T} \rightarrow \mathcal{P}(\mathbb{F})$ identifies the set of flows where the specified node $v \in \mathbb{V}$ and packet class $t \in \mathbb{T}$ appear as the first two elements of the list modeling those flows.

$$\phi(v, t) = \{[v, t, v_{a1}, t_{a1}, ...], [v, t, v_{a2}, t_{a2}...], ...\} \quad (3)$$

To clarify the traffic flow model, a simple example is discussed by exploiting the simple chain represented in Fig. 1, where m_1 is a NAT, m_2 is a firewall, m_3 is a load balancer. Let us suppose the following IP addresses for the chain nodes: 192.168.0.2 for s, 220.120.45.3 for m_1, 150.76.20.9 for m_2, 23.172.10.3 for m_3 and 23.172.10.2 for d. Then, let us consider a traffic flow f from s to the TCP destination port 110 of d. Each element of the chain will receive a different network traffic belonging to this flow, because of active functions – i.e., NAT and load balancer – which can modify the IP addresses.

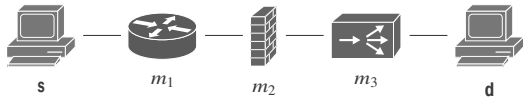


Fig. 1: Chain example to discuss traffic flow model

In particular, the traffic that is generated by s has the IP address of the load balancer as a destination, while the traffics received by the load balancer and by d have the IP address of the NAT as a source. Here are some examples about how the functions beforehand described work for this flow.

$$\pi(f) = [s, m_1, m_2, m_3, d]$$
$$\tau(f, m_1) = t_1, \tau(f, m_2) = t_2, \phi(s, t_1) = \{f\}$$
$$P_{t_1} = \{\textbf{IPSrc} = 192.168.0.2, \textbf{IPDst} = 23.172.10.3, \textbf{pSrc} \subseteq *, \textbf{pDst} = 100, \textbf{tProto} = \textbf{TCP}\}$$
$$P_{t_2} = \{\textbf{IPSrc} = 220.120.45.3, \textbf{IPDst} = 23.172.10.3, \textbf{pSrc} \subseteq *, \textbf{pDst} = 100, \textbf{tProto} = \textbf{TCP}\}$$

Note that, although the packet classes t_1 and t_2 are characterized by different predicates for the same packet fields or attributes, they belong to the same flow.

At this stage, given the formal models of a chain and of its traffic flows, let us introduce the deny predicate:

$$\textbf{deny}: \mathbb{V} \times \mathbb{T} \rightarrow \mathbb{B}$$

where $\mathbb{B}$ is the set of the two possible Boolean values. This predicate is false for a function $v \in \mathbb{V}$ and packet class $t \in \mathbb{T}$ if v can send packets of t to the next element in the chain, while it is true otherwise. The truth of this predicate depends on the behavior and configuration of the function, but it may also depend on the state of the function (e.g., for stateful functions). It is computed by solving the SMT problem, as it will be detailed in Section V.

V. Network Function Modeling

This section describes how the behavior of network nodes is modeled. In particular, since the aim of the verification process is to check the possibility that a specific packet class can reach the destination through a chain, which in turn only depends on the forwarding behavior of each function in the chain, modeling the forwarding behavior is enough. This means that such models must let us know if the modeled function drops or forwards a traffic that belongs to an atomic flow. Note that all packets belonging to an atomic flow are treated by the function in the same way by definition of atomic flow.

A large number of functions is considered in our approach, in order to enable the verification of reachability policies in realistic networks, whose characteristic is the heterogeneity of functions that are exploited to provide a full service to the users. Some examples are packet filter firewalls, web-application firewalls, deep packet inspectors which block undesired communications; VPN gateways and tunnel terminators which enforce security properties such as authentication and integrity; intrusion detection systems which identify potential in-going attacks and consequently alert about their presence; network address translators which shadow an internal network from the outside; load balancers which decide the destination server for any received packet.

The network functions can be classified into two main classes: stateless and stateful functions.

Stateless functions take decisions based on the single analyzed packet without considering previously received traffic. This kind of function is, in fact, not characterized by a state; consequently, it is not able to evaluate conditions related to packets that have previously crossed the function itself. The configuration of a stateless function $m \in \mathbb{M}$ is thus modeled as a pair (R_m, d_m), where R_m is a set of <match, action> rules, whereas d_m is the default action that is applied whenever an incoming packet does not match the conditions of any rule in R_m. In particular, the rules in R_m are disjunct and contiguous, such that each $r \in R_m$ is modeled as $r = (C_r, a_r)$, where C_r is the condition set of the rule, and a_r is the action that must be applied when all the conditions are satisfied. Each condition $c \in C_r$ is modeled as the tuple $c = (p(X), \textbf{attribute})$, where $c.p(X)$ is a predicate that is applied on a set of variables X. Instead, $c.$attribute identifies the packet field or attribute of interest for the predicate and consequently the related variables, already used for the predicates modeling the packet classes, on which it is defined. Then, the actions that must be considered for modeling the forwarding behavior of a network function are $allow$, if the matching traffic can be forwarded, and $deny$, if instead the matching traffic must be blocked. The possibility that a network function might modify an input traffic is, instead, already considered when the flows are computed.

Given an incoming traffic $t \in \mathbb{T}$, a network function $m \in \mathbb{M}$ checks if the traffic fields of t match the conditions of its rules. The result of this check is represented by the match: $R_m \times \mathbb{T} \rightarrow \mathbb{B}$ predicate, which is true for a rule and traffic if the packet fields of the traffic satisfy all the conditions of the rule, false otherwise. (4) shows how the match predicate is formally defined over a rule $r \in R_m$ and an input packet class $t \in \mathbb{T}$.

$$\textbf{match}(r, t) := \forall c \in C_r. \; (\exists p_t \in P_t. \; (c.\textbf{attribute} = \alpha(p_t) \wedge c.p(\xi(p_t)) = \textbf{true})) \quad (4)$$

It is worth underlining that the <match, action> model of a stateless function can be automatically extracted from an abstract representation of a given network function in an imperative language, pursuing approaches such as the one proposed in [23].

As example, the model of a stateless packet filter firewall will be presented in greater detail in Subsection V-A, with some indications about how its model can be modified to represent a deep packet inspector.

For what concerns stateful functions, they take decisions depending not only on each single packet, but considering also the previous packets that arrived at the functions themselves. In order to model state, we should represent time or time-ordering relations. However, this naive approach, followed by other previously proposed methodologies, complicates the formal model. Our alternative approach consists of modeling just the possibility that certain packet classes are forwarded or dropped, without keeping track of time relationships. This

statement will be clarified when the models of some typical stateful functions will be illustrated later in this section. The first is the model of the network address translator, which can allow a communication towards an internal shadowed network only if a corresponding state has been previously set. The second model is, instead, related to a stateful packet filter firewall which combines the configuration of a stateless packet filter with state-based decisions.

A. Examples of stateless functions

Stateless packet filter firewall

An example of a function that belongs to the class of stateless filtering functions is the packet filter firewall, which takes forwarding decisions by considering only the IP 5-tuple of a packet. The packet filter type that is considered in this example is a firewall that works in two different operative modes: (i) whitelisting, when it is configured with a whitelist representing the only kinds of traffic allowed; (ii) blacklisting, when it is configured with a blacklist representing the only kinds of traffic denied. This aspect is modeled with the whitelisting: $\mathbb{V} \rightarrow \mathbb{B}$ predicate, which is true for stateless firewall $m \in \mathbb{V}$, if its operative mode is whitelisting, false otherwise.

Then, for any $r \in R_m$, the A_r list only contains a single action, that is the opposite of the default action: *allow* if R_m is a whitelist, *deny* if it is a blacklist. In the condition set C_r, instead, the predicates that characterize it are defined over the variables of the input packet classes representing the fields of the IP 5-tuple: IPSrc, IPDst, pSrc, pDst, tProto. Since the variables representing addresses or ports can assume interval values, then the predicates that can be defined over them as rule conditions can also check the inclusion of a range in a bigger one.

To show how the match predicate works in this case, a simple example is presented. Let us suppose that a whitelisting packet filter firewall is characterized by the following two rules:

$$r_1 = (\{\textbf{IPSrc} \subseteq \textbf{220.12.5.0/24}, \textbf{IPDst} = \textbf{33.150.10.3}, \textbf{pSrc} \subseteq *, \textbf{pDst} = 80, \textbf{tProto} = \textbf{TCP}\}, \{\textbf{allow}\})$$
$$r_2 = (\{\textbf{IPSrc} = \textbf{220.12.5.1}, \textbf{IPDst} \subseteq \textbf{33.150.0.0/16}, \textbf{pSrc} \subseteq *, \textbf{pDst} \subseteq *, \textbf{tProto} = \textbf{TCP}\}, \{\textbf{allow}\})$$

Let us then consider the following input packet class for the firewall:

$$t = \{\textbf{IPSrc} = \textbf{220.12.5.1}, \textbf{IPDst} = \textbf{33.150.10.3}, \textbf{pSrc} \subseteq *, \textbf{pDst} = 110, \textbf{tProto} = \textbf{TCP}, \textbf{domain} = \textbf{"google.com"}\}$$

It follows that $\textbf{match}(r_1, t) = \textbf{false}$ because the predicates over the destination port are in conflict, whereas $\textbf{match}(r_2, t) = \textbf{true}$ because all the predicates match.

Having presented how the model of the configuration is explicated for a packet filter firewall $m \in \mathbb{M}$, the remaining aspect that must be considered is to model how this function type decides if an input packet class is forwarded or dropped. This is achieved by introducing, for any possible input traffic $t \in \mathbb{T}$, the hard constraints represented in (5).

$$\begin{aligned} &\textbf{deny}(m, t) \iff (a) \vee (b) \\ &(a) = \textbf{whitelisting}(m) \wedge (\nexists r \in R_m.\ \textbf{match}(r, t)) \\ &(b) = \neg\textbf{whitelisting}(m) \wedge (\exists r \in R_m.\ \textbf{match}(r, t)) \end{aligned} \quad (5)$$

In words, m blocks a traffic t in two possible scenarios: (i) m has a whitelisting configuration, without any *allow* rule whose conditions are satisfied by t; (ii) m has a blacklisting configuration, with a *deny* rule whose conditions are satisfied by t.

Deep packet inspector

Another example of a stateless function is the deep packet inspector (DPI). This security function is able to inspect and detect the presence of elements contained in the IP payload. The model for the configuration of a DPI $m \in \mathbb{M}$ can be derived from the packet filter's model, by redefining only the condition set for each rule $r \in R_m$. If for the packet filter predicates on the variables representing the IP 5-tuple are defined, for the DPI the contains predicate is used on the variable representing the IP payload. In this way, when a traffic $t \in T$ is received, the DPI checks if it satisfies the conditions of at least a rule. In that case, the rule's action (i..e, *allow* or *drop*) is applied; otherwise, the default action is enforced.

Like for the packet filter firewall, we present a simple example to clarify how the match predicate works with the DPI's model. Let us consider the following two rules of a blacklisting DPI and the following input packet class:

$$r_1 = (\{\textbf{contains}(\textbf{body}, \textbf{"weapon"})\}, \{\textbf{deny}\})$$
$$r_2 = (\{\textbf{contains}(\textbf{body}, \textbf{"video"})\}, \{\textbf{deny}\})$$
$$t = \{\textbf{IPSrc} = \textbf{220.12.5.1}, \textbf{IPDst} = \textbf{33.150.10.3}, \textbf{pSrc} \subseteq *, \textbf{pDst} = 110, \textbf{tProto} = \textbf{TCP}, \textbf{contains}(\textbf{body}, \textbf{"weapon"})\}$$

It follows that $\textbf{match}(r_1, t) = \textbf{true}$, whereas $\textbf{match}(r_2, t) = \textbf{false}$.

Note that these models can be extended in order to also consider filtering functions that can modify the received packets before forwarding them to the next hop.

B. Examples of stateful functions

Network address translator

A first example of stateful function is the network address translator (NAT). It shadows the private IP addresses of the end-hosts of a sub-network by replacing them with its own address and, at the same time, it filters traffic coming from outside if it does not represent a reply to a communication started from the shadowed sub-network. To characterize this behavior, considering a NAT $m \in \mathbb{M}$, the I_m set contains the IP addresses assigned to any shadowed end-host that can open communications towards external networks. Note that, since end hosts can be subnetworks, each $i \in I_m$ can be not only a specific address, but also a range or a set of addresses as well.

The predicate $\textbf{shadowed}: \mathbb{M} \times \mathbb{T} \rightarrow \mathbb{B}$ thus models how a NAT recognizes an IP address that must be shadowed. This predicate returns true if the value hold by the variables representing the source IP address for the packet class $t \in \mathbb{T}$ is included in at least an element $i \in I_m$. This behavior can be defined as shown in (6).

$$\textbf{shadowed}(m, t) := (\exists i \in I_m.\ \exists p \in P_t.\ \alpha(p) = \textbf{"IPSrc"} \wedge \xi(p) \subseteq i) \quad (6)$$

The forwarding behavior of a NAT is, then, modeled with a pair of constraints. The first, shown in (7), states that a NAT $m \in \mathbb{M}$ can forward the traffic $t \in \mathbb{T}$ to the next

hop if the source IP address is shadowed by the NAT and consequently belongs to an end-host which can open any communication towards external networks. For this kind of packet class, the source IP address is modified with one of the addresses assigned to the NAT.

$$\textsf{shadowed}(m,t) \implies \neg\textsf{deny}(m,t) \tag{7}$$

The second constraint, instead, is related to the forwarding of traffic whose source IP address is not shadowed by the NAT. In this case, the traffic can be forwarded if there exists the possibility that a packet class with opposite characteristics can be received by the NAT. Through this definition, we can avoid to introduce a variable representing the time, because the verification process does not check that the packet class has been effectively received in the past, but only the possibility that it may be received.

For modeling this second constraint, three notations are introduced:

- $\mu\colon \mathbb{V}\times\mathbb{T}\to\mathbb{T}$ identifies the output traffic $t_2\in\mathbb{T}$ that is generated by the node $v\in\mathbb{V}$, after it has received the input traffic $t_1\in\mathbb{T}$.
- opposite: $\mathbb{T}\times\mathbb{T}\to\mathbb{B}$ checks if two traffic elements $t_1\in\mathbb{T}$ and $t_2\in\mathbb{T}$ have opposite characteristics. Considering that the NAT works at level 3-4 of the stack ISO/OSI, then the opposite predicate analyzes and compares only the variables related to the IP 5-tuple, checking that the IP address and port source of a traffic t_1 is equal to the corresponding destination field of the traffic t_2.
- given a chain $c\in\mathbb{C}$ and two network nodes $m_i, m_j\in c.M$ such that $m_i\neq m_j$, $m_i\prec m_j$ means that m_i precedes m_j in the chain.

Having introduced these notations, the forwarding of packet classes coming from external (i.e., not shadowed) end-hosts is modeled with the constraint (8).

$$\begin{aligned}\neg\textsf{shadowed}(m,t) \implies (\textsf{deny}(m,t) \iff (\nexists f\in\mathbb{F}.\,(a)\wedge(b)))\\ (a)=\textsf{shadowed}(m,\tau(f,m))\wedge\ \textsf{opposite}(\tau(f,m),\mu(m,t))\\ (b)=\forall v\in\pi(f)\mid v\prec m.\,(\neg\textsf{deny}(v,\tau(f,v)))\end{aligned} \tag{8}$$

In words, a traffic coming from the outside can be forwarded by the NAT if it exists the possibility that a traffic coming from a shadowed end-host with opposite characteristics can be received (i.e., there is not any middle-box that, in the path from the source to the NAT, would block it).

Stateful packet filter firewall

A second example is the stateful packet filter firewall. The version that has been modeled is a whitelisting firewall, with a *deny* default action and a set of rules whose action is to allow any traffic which satisfies the rule conditions. The difference is, however, that every time a different kind of traffic is received, the related information is internally stored and represents the firewall's state. Consequently, when an input traffic does not satisfy the conditions of any rule, before applying the default action, the function checks if traffic with opposite characteristics has been received. In that case, the communication is allowed.

The model of a stateful packet filter firewall $m\in\mathbb{M}$ is a combination of the stateless firewall's and NAT's models. First, focusing on the pair (R_m, d_m), d_m is equal to *deny*, while the condition set for every $r\in R_m$ is defined with the same predicates as for the stateless version. The match predicate works in the same way as well. Then, the forwarding behavior of the firewall, given an input traffic $t\in\mathbb{T}$, is modeled with the constraints (9).

$$\begin{aligned}\textsf{deny}(m,t) \iff (\nexists r\in R_m.\,\textsf{match}(r,t))\wedge(\nexists f\in\mathbb{F}.\,(a)\wedge(b))\\ (a)=\textsf{opposite}(\tau(f,m),\mu(m,t))\\ (b)=\forall v\in\pi(f)\mid v\prec m.\,(\neg\textsf{deny}(v,\tau(f,v))\wedge\\ (\nexists r\in R_m.\,\textsf{match}(r,\tau(f,m))))\end{aligned} \tag{9}$$

In words, an input traffic t is dropped by the stateful packet filter m if and only if there is no rule of m that matches t and there is no flow f passing through m such that (b) the ingress traffic of m belonging to f is opposite to the traffic that m outputs when it receives t, and (c) there is no function belonging to f before m that drops the traffic of f.

In a similar way, other stateful functions can be modeled and supported by our approach, introducing additional constraints in the function model.

VI. Reachability Modeling

This section introduces the formal models of reachability and reachability policy, and the verification algorithm.

A. Reachability formal definition

As our main aim is to formally verify reachability between end-hosts of a network, i.e. between the elements of $\mathbb{H}$, a formal notion of the reachability concept is needed.

Reachability for a single chain $c=(s,d,M)$ and traffic t is expressed by the reach predicate:

$$\textsf{reach}\colon \mathbb{C}\times\mathbb{T}\to\mathbb{B}$$

Intuitively, $\textsf{reach}(c,t)$ is true if, assuming the source node $c.s$ of the chain generates traffic t, targeted to destination node $c.d$, t can actually reach its destination. Equation (10) shows how this predicate is formally defined.

$$\textsf{reach}(c,t) := (\exists f\in\phi(c.s,t).(\forall m\in c.M.\,\neg\textsf{deny}(m,\tau(f,m)))) \tag{10}$$

In words, $\textsf{reach}(c,t)$ returns true when there exists the possibility that a flow f whose first two elements are $c.s$ and t can reach the destination $c.d$ through the chain c. It is indeed possible that the traffic is modified by intermediate middleboxes such as NATs or load balancers. Concerning these possible modifications, the model that we have introduced for the traffic flows is sufficiently expressive to consider the possibility that different packet classes can belong to the same end-to-end communication from $c.s$ to $c.d$.

B. Reachability policy model

Reachability policies are used by administrators to express the reachability-based requirements that they want to verify in a network graph. Formally, a reachability policy p is a tuple, composed by a source s, a destination d, a packet class t generated by s and a restriction r.

$$p=(s,d,t,r)$$

First of all, policy p expresses Assertion 1:

Assertion 1.
"The traffic t, generated from the source s, can reach the destination d respecting the restriction r"

Specifically, r is a tuple composed of (i) a *type*, which can be *none*, *selection*, *set*, *sequence* or *list*; and (ii) a set of service functions M which the specified traffic must pass through.

$$r = (type, M)$$

When the administrator does not want to specify any restriction for a policy p, then $r.type$ is set to *none* and $r.M$ is the empty set $\emptyset$. Otherwise, when $r.type \neq none$, Assertion 2 must be true:

Assertion 2.
"The flows whose first two elements are s and t respectively must pass through all the functions in M"

Moreover, if the restriction type is equal to *sequence* or *list*, this means that M is an ordered set. In this case, Assertion 3 must also be true:

Assertion 3.
"The flows whose first two elements are s and t respectively must sequentially pass through all functions in M"

Whereas, if the administrator specifies that the type of r is equal to a *set* or a *list*, this means that the administrator wants the packets derived from t to pass through only the functions specified in M. In this case, Assertion 4 must be true:

Assertion 4.
"The flows whose first two elements are s and t respectively must pass through only the functions specified in M"

In summary:

- if $type = selection$, Assertion 2 must be true;
- if $type = set$, Assertions 2 and 4 must be true;
- if $type = sequence$, Assertions 2 and 3 must be true;
- if $type = list$, Assertions 2, 3 and 4 must be true.

The predicate stf*(r,c)* is used to specify if the restriction r is satisfied in the chain c:

$$\textbf{stf}(r, c) := \begin{pmatrix} (r.type = \text{“}selection\text{”} \wedge r.M \backslash c.M = \emptyset) \vee \\ (r.type = \text{“}set\text{”} \wedge r.M \backslash c.M = c.M \backslash r.M = \emptyset) \vee \\ (r.type = \text{“}sequence\text{”} \wedge r.M \subset c.M) \vee \\ (r.type = \text{“}list\text{”} \wedge r.M = c.M) \end{pmatrix} \quad (11)$$

For example, let us consider a policy related to the communication from a group of clients to a server of their company's network. Even though a reachability invariant is necessary, at the same time a restriction is needed to protect the server from potential attacks. A first possibility is to define the restriction as $r = (\textbf{selection}, \text{“}\textbf{deep packet inspector}\text{”})$: in this case, the traffic must at least pass through a function performing a deep inspection and alerting when an attack is detected. A second, more protective possibility is to to define the restriction as $r = (\textbf{sequence}, \text{“}\textbf{firewall, deep packet inspector}\text{”})$: supposing the firewall is already configured to block specific packet classes, then the deep packet inspector is put behind it so that alerts are risen only when the protection provided by the firewall is overcome. Note that, if instead of *selection* and *sequence* the types of r were *set* and *list*, then the deep packet inspector and the firewall would be the only functions which could be crossed between source and destination, forbidding that the traffic passes through other potentially malicious middleboxes.

Finally, it is interesting to note that an administrator may omit the source, destination and packet class, using the symbol '*'. This means that all sources, destinations or packet classes are evaluated.

$$s_i = * \rightarrow \forall \textbf{ source}$$
$$d_i = * \rightarrow \forall \textbf{ destination}$$
$$t_i = * \rightarrow \forall \textbf{ packet class}$$

For example, $p_1 = (A, *, *, r)$ and $p_2 = (*, A, *, r)$, with $r = (none, \emptyset)$, permit to verify if the end-host A can be reached by any other end-host with packets of any class, without any other restriction.

C. Reachability verification

In order to enable the verification of the requirements defined by the administrator as reachability policies, let us introduce the predicate enforced(p, c), which is true if policy p is correctly enforced in chain c. This predicate can be expressed in terms of the previously defined predicates as follows.

$$\textbf{enforced}(p, c) := \textbf{stf}(p.r, c) \wedge \textbf{reach}(p.t, c) \quad (12)$$

Having defined predicate enforced(p, c), we define $\chi(p, \mathbb{C})$ as the function that identifies all chains in $\mathbb{C}$ that correctly enforce the policy p.

$$\chi(p, \mathbb{C}) := \{c \in \mathbb{C} | \textbf{ enforced}(p, c)\} \quad (13)$$

We present other five predicates that allow us to introduce additional requirements about how many chains need to satisfy the requirements specified in a policy. Specifically, these predicates allow an administrator to verify if *at least one*, *only one*, *more than one*, *all* or *no* chains satisfy the requirements, thus allowing many varations (including, e.g., isolation verification).

Predicate enforced$_{one}(p, \mathbb{C})$ verifies if at least one chain exists that enforces the requirements in the policy p:

$$\textbf{enforced}_{one}(p, \mathbb{C}) := |\chi(p, \mathbb{C})| \geq 1 \quad (14)$$

Predicate enforced$_{only}(p, \mathbb{C})$ verifies if only one chain exists that enforces the requirements in the policy p:

$$\textbf{enforced}_{only}(p, \mathbb{C}) := |\chi(p, \mathbb{C})| = 1 \quad (15)$$

Predicate enforced$_{more}(p, \mathbb{C})$ verifies if more than one chain exist that enforce the requirements in the policy p:

$$\textbf{enforced}_{more}(p, \mathbb{C}) := |\chi(p, \mathbb{C})| > 1 \quad (16)$$

Predicate enforced$_{all}(p, \mathbb{C})$ verifies if all chains enforce the requirements in the policy p:

$$\textbf{enforced}_{all}(p, \mathbb{C}) := |\chi(p, \mathbb{C})| = |\mathbb{C}| \quad (17)$$

Predicate enforced$_{none}(p, \mathbb{C})$ verifies if no chains enforce the requirements in the policy p:

$$\textbf{enforced}_{none}(p, \mathbb{C}) := |\chi(p, \mathbb{C})| = 0 \quad (18)$$

Moreover, we define two other functions handy to identify which chains do not respect a policy (χ_{others}) and which

function in a chain blocks the traffic specified in the policy (β). Given $m \in c.M$, let us denote c_1 the subchain extracted from c where the source is $c.s$ and the destination is m, c_2 the subchain extracted from c where the source is m and the destination is $c.d$. Then:

$$\chi_{others}(p, \mathbb{C}) := \{\mathbb{C} \setminus \chi(p, \mathbb{C})\} \quad (19)$$

$$\beta(p, c) := \{m \in c.M \mid \mathsf{reach}(p.t, c_1) \wedge \neg \mathsf{reach}(p.t, c_2)\} \quad (20)$$

D. Reachability process

After the introduction of the predicates that are needed for the verification of the reachability policies, the process by means of which their formal verification is effectively performed is described in the following.

Firstly, given a reachability policy $p = (s, d, t, r)$, only the chains on which it must be checked are extracted. A chain $c \in \mathbb{C}$ such that $c = (s, d, M)$ is extracted for p if $c.s = p.s$ and $c.d = p.d$.

Secondly, for each chain c extracted in the previous stage, the traffic flows for that chain that start with the source node and traffic specified in the policy are computed. This computation is based on the traffic transformation made by each function in the chain and on the atomic flow definition provided in Section IV. This is feasible by analyzing the configuration of the intermediate functions, focusing on their transformation actions (e.g., a NAT modifies a packet coming from an internal network replacing the source IP address with an external address) and overlooking their forwarding behavior (i.e., if they drop or not an input traffic).

At this point, after the computation of the flows, the SMT problem is formulated. In this problem instance, the constraint described in (10) is introduced to impose the enforcement of the reachability property in the chain. Then, all the other constraints which might generate some conflicts with this one are introduced. These constraints involve the formal model of network and traffic flows. In particular, in this stage only the forwarding behavior is considered for the network functions, because the way each function modifies input packet classes has been already addressed in the previous step. Examples of constraints modeling the forwarding behavior of the network functions, and having an impact on the *deny* predicate, are (5) for a stateless firewall, (7) and (8) for a stateful NAT.

After solving the problem instance, if the outcome is positive, this means that the reachability has been formally verified for the policy on an extracted chain. Alternatively, a negative outcome means that the source of the policy cannot reach the destination by generating the traffic specified by the policy itself.

The same applies to all the other chains, possibly in parallel. This way, it is also possible to check if at least one, only one, more than one, all or no chains satisfy the policy, according to the previously described purposes.

VII. Model Validation

In this section, we first introduce the tool we developed, implementing the proposed approach. Then, we provide an extensive evaluation for both the approach and the implementation. Finally, we compare our technique with two of the most relevant state-of-the-art tools and with the previous version of VeriGraph.

A. Implementation

VeriGraph2.0 is an evolution of the previous framework, VeriGraph, based on the new verification approach proposed in this paper. It is a Java-based framework, released as open source tool[3] under the AGPLv3 license. Users can interact with this framework by means of proper REST APIs. For example, they can provide the service graph description in a custom JSON, YAML, or XML-based format.

In order to exhaustively verify all the possible service chains available in a given service graph, VeriGraph2.0 exploits the Neo4j APIs [24] to store the graph on a local database and to extract all the chains for a given source-destination pair. Neo4j is a graph database platform, with a highly flexible internal structure based on the so-called "stored relationships" between data records. Each record is, in fact, characterized by direct pointers to all the other records with which it has a relationship. This innovative storage structure is exploited by the query language that is used in Neo4j, where there is no need to compute relationships because they are already stored. Neo4j has been selected for these reasons, alongside with its generality and simplicity. We acknowledge other database solutions (e.g., OrientDB) or optimized path extraction algorithms might be considered as alternative. However, a comparison among existing solutions in this field is outside the scope of this paper, whose focus is on the efficiency of the formal verification methodology. For this reason, this analysis is left for future work.

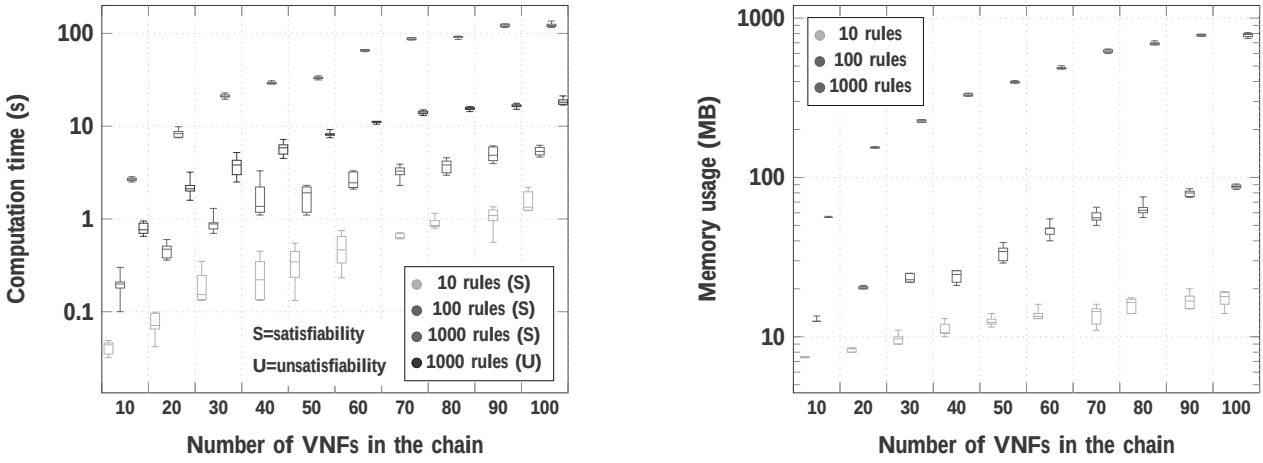
On the extracted chains, VeriGraph2.0 performs (possibly in parallel) the formal verification process. For this purpose it exploits Z3, a state-of-the-art theorem prover developed by Microsoft Research [25]. This tool can be used to model and solve different kinds of mathematical logic and decision problems, from traditional SAT problems to more complex variants such as SMT. Therefore, it supports many theories and offers the versatility that is required for the definition and solution of complex problems, such as the verification problem described in this paper.

B. Evaluation

An extensive evaluation of the proposed approach has been carried out considering both the accuracy and the performance for the two stages of the approach (i.e., the chain extraction and the following reachability verification on each chain). This evaluation has been performed on a 4-core Intel i7-6700 3.40 GHz workstation with 32 GB RAM. The adopted Z3 version is 4.8.5, while the adopted version for Neo4j is 3.1.3.

Regarding the accuracy property, the correctness of the chain extraction is intrinsic in the use of Neo4j as graph database platform, which is a well-known and worldwide adopted tool for operating on graphs. Instead, the correctness

[3] https://github.com/netgroup-polito/verigraph



(a) Computation time chart

(b) Memory usage chart

Fig. 2: Impact of the length of chains and policy on computation time and memory.

of the reachability verification performed on each chain has been evaluated in different ways, considering both synthetically generated network chains and realistic use cases later presented in this section (e.g., in Subsections VII-D and VII-E). More specifically, some configuration errors have been introduced for the functions composing the chain so that the reachability between the end points of the chain is no longer guaranteed (e.g., a filtering rule is added in an intermediate firewall to block the traffic). We have thus checked that the reachability property is satisfied when the configuration error is absent, not satisfied when the error is present. Furthermore, whenever possible, we have applied other verification tools (e.g., Symnet [6]) on the same use cases, verifying that their verification results are coherent with ours.

Regarding the performance property, the chain extraction has been evaluated by generating a number of random service graphs of different sizes and calculating the time spent to extract the chains from them. The outcome of this first analysis is that the time spent on chain extraction is proportional to the size of the graph, but in any case satisfactory for real scenarios. For example, it takes around 1 second for a graph containing 80 nodes. In contrast, the time taken by reachability evaluation depends on many factors, e.g., the length of the extracted chain, and the type and number of rules in the configuration of each network function in the chain. For this reason, in the rest of this section we present an evaluation of this stage, by considering a single chain with varying features.

The scalability assessment of the reachability verification is instead based on a synthetically generated service chain, that is built according to the following conditions:

- 70% of the chain length is made by security filtering functions (40% packet filter firewalls, 15% web-application firewalls and 15% deep packet inspectors);
- 30% of the chain length is made by other network functions (network address translators, load balancers, traffic monitors and forwarders).

The ratio of network functions constituting the NFV chain has been chosen to be 7:3 because it represents an unfavorable, yet not totally unrealistic, case for the approach illustrated in this paper, with respect to the ratios typically found in real networks.

Each type of chain is thus identified by: (i) total number of VNFs in the chain; (ii) number of filtering rules in the configuration of each VNF representing a security function. For each type of chain, we evaluate the computation time of the verification process. Fig. 2a plots the results that have been achieved for different types of chain, over 100 iterations for each different test case. The box plots depicted in Fig.2a show minimum, 5th percentile, median, 95th percentile and maximum for the computed results. The same applies to all the other box plots in the next figures of this section. For the results depicted Fig. 2a, a differentiation with respect to the output of the tests is made. More specifically, the box plots associated to the S letter (with S standing for satisfiability) are related to tests where it is verified that the reachability property is satisfied, while the box plots associated to the U letter (with U standing for unsatisfiability) are related to tests where the reachability property is not satisfied. This differentiation has no relevance for Fig. 2b, because it only impacts the computation time.

A first observation is that the verification time does not show an exponential behavior with respect to both the chain length and the number of rules in each security function. This result proves the scalability of our methodology and its applicability to real network scenarios. For example, in the worst case we considered in this validation, the computation time required to perform the verification of a reachability invariant is 121 seconds, considering a chain made by 100 VNFs with 1000 rules in each security VNF.

Another observation is that not only the chain size, but also the number of rules, has an impact on the performance, as expected. This is particularly evident if the boxes representing the cases of 100 rules and 1000 rules in Fig. 2a are compared. Each filtering rule is actually modeled as a constraint in the verification problem and the Z3 solver consequently needs additional time to parse this constraint, together with all the

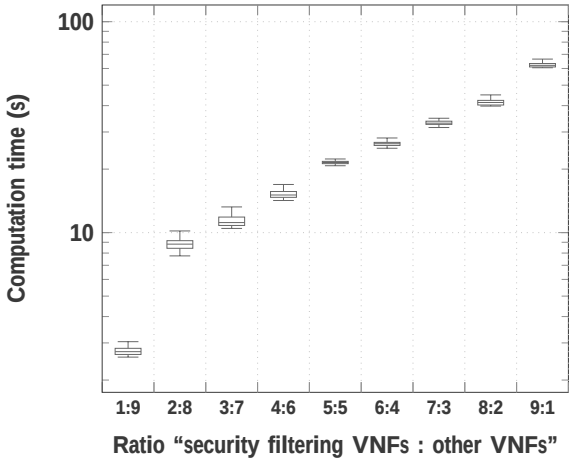


Fig. 3: Impact of network function types on computation time

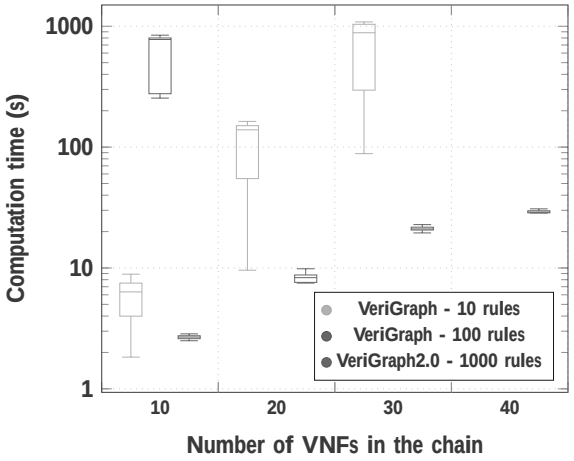


Fig. 4: Comparison between VeriGraph2.0 and VeriGraph

others. However, it is worth noticing how the 1000 rules case represents a very extreme scenario, where 70% of all the VNFs in each chain are security functions configured with 1000 rules each. Hence, the obtained computation time in this case, although large with respect to the other cases, is a further demonstration of the effectiveness of the approach in terms of scalability.

It is also interesting to observe the difference in performance between scenarios where the reachability invariant holds (blue boxes) and scenarios where the reachability property is not satisfied (black boxes). The verification time required in the latter case is lower than the time needed in the former, even though most of the constraints are the same. The reason is that the adopted solver, Z3, is able to really fast identify a conflict between the constraints, whereas it takes more time to check if all of them can be satisfied.

The peak memory usage of the framework for different number of VNFs in the chain is then shown in Fig. 2b. As it can be observed, the required memory space in the scenarios where each security VNF is configured with 1000 rules is much higher than the other cases. The reason is the presence of a bigger number of constraints and variables representing the rules. Nevertheless, the peak memory usage does not exceed 800 MB even in the worst analyzed scenario, which is again an extreme case of a chain made by 100 VNFs with 1000 rules in each security VNF.

The results presented so far have been achieved by keeping the ratio between the number of security filtering VNFs and other VNF types fixed at 7:3. Instead, Fig. 3 depicts the outcome of an experimental evaluation, where this ratio is varied from 1:9 to 9:1. The chain subject to this evaluation is composed of 50 VNFs, and each security VNF is configured with 1000 rules. A higher time is required for the verification process as the percentage of security filtering VNFs increases in the chain. This result is expected, because the formal models designed for that class of functions are more complex than the models of other VNF types. However, even the results deriving from a 9:1 ratio, which represents an extreme scenario, are in the same magnitude order as the ones deriving from the 7:3 ratio, used for the scalability validation of VeriGraph2.0.

The achieved results overall show that the verification time is in the order of seconds in the case of real-sized network scenarios. This is reasonably in line with the requirements dictated by an SDN/NFV environment, especially in terms of time required by the verification process to authorize a newly asked network reconfiguration. Moreover, the achievement of such a scalability level has not been shown by other existing works related to the formal verification of reachability invariants. With this respect, in the rest of this section we compare VeriGraph2.0 with its predecessor and with two of the most relevant state-of-the-art tools for reachability verification, in order to further support the effectiveness of our approach.

C. VeriGraph2.0 vs VeriGraph

As already explained, VeriGraph2.0 is the evolution of VeriGraph [13]. Even though the tools used to implement VeriGraph2.0 are the same used for VeriGraph (i.e., Neo4j for chain extraction, and Z3 for policy verification), the formulation of the SMT problem is completely different: as detailed in the previous sections, Verigraph2.0 models packet flows by means of predicates, rather than modeling individual packets, and this model also enabled the elimination of quantifiers. Moreover, Verigraph2.0 uses formal models of network functions that are more detailed, so representing the function behavior more closely to reality. For example, VeriGraph2.0 models the rule conditions of a packet filter firewall as predicates over the fields of the IP 5-tuple, while in Verigraph they were simply defined over the IP source and IP destination fields. Additionally, the policy model is richer; for instance, as illustrated in Section VI, in VeriGraph2.0 it is possible to define restrictions for the enforcement of reachability policies, which were not possible in Verigraph.

For what concerns performance, Fig. 4 shows the results of a comparison between the two frameworks. The tests have been carried out under the same conditions explained for the tests relative to Fig. 2a, where a reachability property is positively verified on a chain of increasing length. As it is clear from this chart, the results achieved by VeriGraph2.0 when each security VNF of the chain is configured with 1000 rules are much better than the results achieved by VeriGraph on a chain of the same

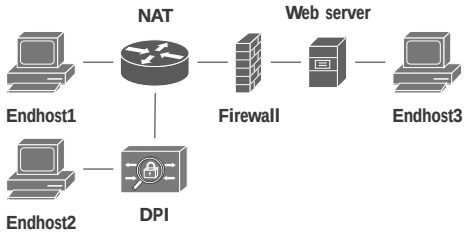


Fig. 5: Topology used in the comparison with SymNET

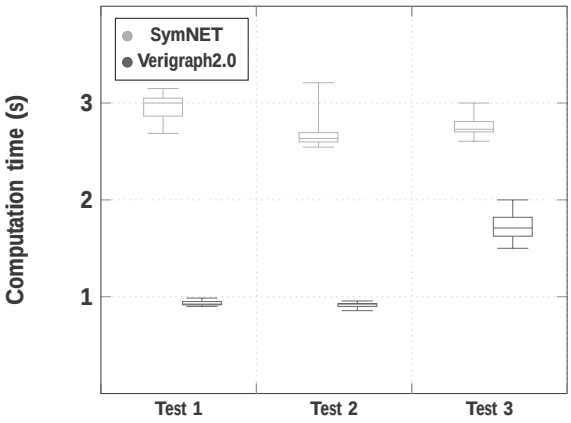


Fig. 6: Comparison between VeriGraph2.0 and SymNET

length, but where each security VNF is configured with only 10 or 100 rules. Moreover, VeriGraph was not able to reach a solution for the SMT problem in the following cases: chains of length higher than 30 with security VNFs configured with 10 rules; chains of length higher than 10 when they are configured with 100 rules; all the analyzed chains (including those that are 10 VNFs long) when they are configured with 1000 rules. Therefore, the improvement reached by VeriGraph2.0 is evident with respect to its predecessor: it can be successfully applied with better performance and on longer chains, having VNFs with more rules.

D. VeriGraph2.0 vs SymNET

In this section, we compare our approach with SymNET [6], one of the latest tools for network reachability verification available in the literature. As briefly described in Section II, SymNET analyzes an abstract data plane model and checks network configurations by using symbolic execution. A major difference with respect to VeriGraph2.0 is that it explores the entire network within a single execution, whichever is the number and the type of policies to verify. VeriGraph2.0, instead, extracts the chains and then considers only relevant paths for the policies to verify, possibly in parallel.

The service graph adopted in this comparison, shown in Fig. 5, includes three end-hosts and some representative network elements (NAT, firewall, DPI, web server). Using the tools, we perform a reachability analysis on the given graph, considering different test case scenarios by changing the configurations for the various network elements. Fig. 6 shows the box plot for the total time that is required by VeriGraph2.0 and SymNET for the entire network exploration (it includes chain extraction in VeriGraph2.0) and verification, averaged over 100 runs for each test case.

In Test 1, we configure the firewall to drop all the packets coming from the NAT to the web server and we perform the reachability analysis from End-host 1 to the web server. In Test 2 instead, the firewall is set to block only packets coming from End-host 1 towards the webserver and the DPI is set to block packets containing the string “virus” . In this scenario, we check if a flow not containing the string “virus” in its body will reach the Web Server from End-host 2. Finally, in Test 3 we check the reachability property between the web server and End-host 3, without considering any configuration for the network functions. The results show VeriGraph2.0 being more performant than SymNET, on average, in terms of execution time. In particular, VeriGraph2.0 shows a gain in performance which is around or larger than 50% with respect to SymNET. This could make the verification and then the deployment of new services faster, thus favoring the rise of more flexible networking services. Besides, the outputs produced by both the tools are the expected ones: this further confirms the accuracy of the approach proposed in this paper.

E. VeriGraph2.0 vs VMN

Like Verigraph2.0, VMN [8] can verify reachability policies in computer networks composed of several stateless and stateful functions. Nevertheless, the formal models used to formulate the verification problem are highly different. First of all, the models used by VMN are based on LTL and single packets, whereas VeriGraph2.0 adopts a direct FOL formalization of packet flows, which entails higher simplicity. Moreover, VeriGraph2.0 provides a larger pool of available functions, and richer models for them (e.g., the difference for packet filter models is the same illustrated in the comparison with VeriGraph).

For what concerns performance, a comparison has been made considering a topology illustrated in [8], which we replicated to assess the performance of VeriGraph2.0 in the same network scenario. Since it was not possible for us to access the VMN tool, our results are compared with the ones presented in the paper. The adopted topology (shown in Fig. 7) is a network that interconnects the Internet (represented as an end point of the communications) with other subnets through a firewall and a gateway. The subnets are classified into three different types: public, private and quarantined. Hosts belonging to a public subnet are allowed to initiate and accept connections in both directions, whereas hosts of a private subnet cannot accept incoming connections. Instead, hosts in a quarantined subnet should be node isolated: this means they are even not allowed to open communications to the outside of their subnet. This scenario allows us to perform a comprehensive comparison between the two tools as it includes both reachability and isolation verification problems. Moreover, it is also useful to evaluate the performance of VeriGraph2.0 against the complexity of the defined policies.

The results of the comparison are depicted in Fig. 8. In the x-axis, the network size is the total number of nodes of the graph (i.e., end hosts and middleboxes). For the network

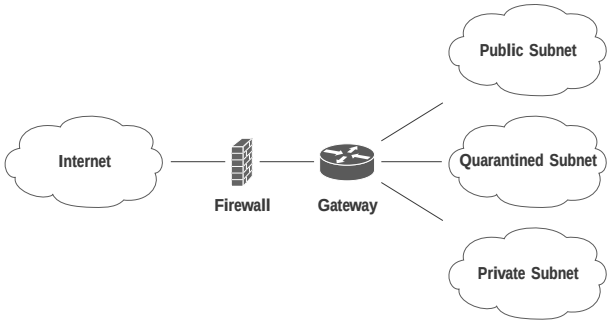


Fig. 7: Topology used in the comparison with VMN

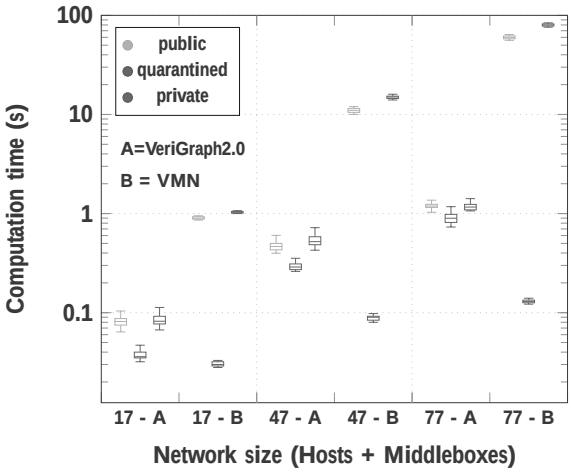


Fig. 8: Comparison between VeriGraph2.0 and VMN

size of each test case, two network nodes are the firewall and the gateway. The remaining nodes are the end hosts, divided into three equally numerous groups: one third is made of public subnets, one third of private subnets, and one third of quarantined subnets. The firewall is configured with a rule for each private subnet, and two rules for each quarantined one.

From the analysis of the results, in the cases of public and private subnets, it is possible to note a significant improvement of performance shown by VeriGraph2.0 with respect to VMN. It is worth noticing that the adopted test environments are not completely aligned (a 10-core Xeon 2.6 GHz processor with 256GB RAM is used in [8]). Since the computational resources of the environment used to test VeriGraph2.0 are inferior to those used to test VMN, the improvement achieved by VeriGraph2.0 is even more than what appears from Fig. 8. Only in the case of quarantined subnets, the performance of VeriGraph2.0 is slightly worse. The reason is that in this scenario we check the unsatisfiability of the reachability policy (i.e., the isolation), which requires a shorter exploration of the space of solutions. Consequently, the difference in terms of modeling between VeriGraph2.0 and VMN has a lower impact and the difference of test environment (e.g., CPU speed, RAM) might be more significant.

In conclusion, even though the results of the tests on VeriGraph2.0 are not always better than those on VMN, they are significantly better in most tests. Considering that VeriGraph2.0 uses richer function models and that the computational resources of its testing environment are inferior, the approach illustrated in this paper shows indeed advantages with respect to the alternative approach pursued in VMN.

VIII. Conclusions

This paper proposes a network modeling and a verification approach, that allows network and security administrators to check reachability-based policies in a service function graph, composed of both stateless and stateful network security functions.

Our reachability verification has a high usability because it supports a large number of network security functions and allows administrators to verify different types of reachability-based requirements. Furthermore, it is totally reusable in case of network topology changes, because we separate the generic forwarding principles from the functional behavior of the existing network functions.

We implemented our model in Java by using Z3, a state-of-the-art Satisfiability Modulo Theory solver, and we validated it in several network scenarios. We also compared our tool with the state-of-the-art solutions in this field, showing how it is able to significantly reduce verification times. This result is obtained by means of efficient modeling and verification techniques, and is fundamental for the dynamic SDN/NFV environment where network topologies and configurations can rapidly change.

For what concerns possible future work, we plan to extend the expressivity of our network model by adding support for new types of network security functions, such as intrusion detection systems. Furthermore, we are planning to extend our approach by supporting other types of verification, related for example to information disclosure, latency constraints, and reliability.

References

[1] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 236–262, 2016.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, April 2014.

[3] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," RFC 7665, Oct. 2015.

[4] P. Quinn and T. Nadeau, "Problem statement for service function chaining," RFC 7498, April 2015.

[5] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 499–512.

[6] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 314–327.

[7] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J. Kang, "Sfc-checker: Checking the correct forwarding behavior of service function chaining," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Palo Alto, CA, USA, November 7-10, 2016*, 2016, pp. 134–140.

[8] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 699–718.

[9] P. Mishra, E. S. Pilli, V. Varadharajan, and U. K. Tupakula, "Intrusion detection techniques in cloud environment: A survey," *J. Netw. Comput. Appl.*, vol. 77, pp. 18–47, 2017.

[10] S. T. Ali, V. Sivaraman, A. Radford, and S. Jha, "A survey of securing networks using software defined networking," *IEEE Trans. Reliab.*, vol. 64, no. 3, pp. 1086–1097, 2015.

[11] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 113–126.

[12] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 290–301.

[13] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, "Formal verification of virtual network function graphs in an sp-devops context," in *Service Oriented and Cloud Computing*. Springer, 2015, pp. 253–262.

[14] W. John, G. Marchetto, F. Nemeth, P. Skoldstrom, R. Steinert, C. Meirosu, I. Papafili, and K. Pentikousis, "Service provider devops," *IEEE Commun. Mag.*, vol. 55, no. 1, pp. 204–211, January 2017.

[15] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10.

[16] 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," *SIGPLAN Not.*, vol. 49, no. 6, pp. 282–293, Jun. 2014.

[17] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 99–111.

[18] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 15–27.

[19] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 953–967.

[20] K. Zhang, D. Zhuo, A. Akella, A. Krishnamurthy, and X. Wang, "Automated verification of customizable middlebox properties with gravel," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 221–239.

[21] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "Netsmc: A custom symbolic model checker for stateful network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 181–200.

[22] D. Monniaux, "A survey of satisfiability modulo theory," in *CASC 2016 - 18th International Workshop, Bucharest, Romania, September 19-23, 2016, Proceedings*, ser. Lecture Notes in Computer Science, vol. 9890. Springer, 2016, pp. 401–425.

[23] G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "A framework for verification-oriented user-friendly network function modeling," *IEEE Access*, vol. 7, pp. 99 349–99 359, 2019.

[24] J. Webber, "A programmatic introduction to neo4j," in *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25, 2012*, G. T. Leavens, Ed. ACM, 2012, pp. 217–218.

[25] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

Daniele Bringhenti received the M.Sc. degree (summa cum laude) in computer engineering from the Politecnico di Torino, Italy, in 2019, where he is currently pursuing the Ph.D. degree in control and computer engineering. His research interests include novel networking technologies, automatic orchestration and configuration of security functions in virtualized networks, formal verification of network security policies.

Guido Marchetto is an associate professor at the Department of Control and Computer Engineering of Politecnico di Torino. He got his Ph.D. in Computer Engineering in April 2008 from Politecnico di Torino. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include network protocols and network architectures.

Riccardo Sisto received the Ph.D. degree in Computer Engineering in 1992, from Politecnico di Torino, Italy. Since 2004, he is Full Professor of Computer Engineering at Politecnico di Torino. His main research interests are in the area of formal methods, applied to distributed software and communication protocol engineering, distributed systems, and computer security. He has authored and co-authored more than 100 scientific papers. He is a Senior Member of the ACM.

Serena Spinoso received her M.Sc.Degree (summa cum laude) and PhD in Computer Engineering in 2013 and 2017 from Politecnico di Torino, Turin, Italy. Her research interests include techniques for configuring network functions in NFV-based networks and formal methods applied to verify forwarding correctness of SDN-based networks.

Fulvio Valenza received the M.Sc. (summa cum laude) and Ph.D. (summa cum laude) degrees in computer engineering from the Politecnico di Torino, Turin, Italy, in 2013 and 2017, respectively. His research interests include network security policies. He is currently a Researcher with the Politecnico di Torino, where he works on orchestration and management of network security functions in the context of SDN/NFV-based networks.

Jalolliddin Yusupov received the M.S. degree in 2016 and Ph.D. in Computer Engineering from Politecnico di Torino, Italy, in 2020. Currently, he is the head of the Academic Department at Turin Polytechnic University in Tashkent, Uzbekistan. His primary research interests include formal verification of security policies in automated network orchestration. His other research interests include modeling, cyber physical systems, and cloud computing systems.