Formally Specifying and Checking Policies and Anomalies in Service Function Chaining

Fulvio Valenza*, Serena Spinoso, Riccardo Sisto

[a]*Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy*

Abstract

One of the proposed management strategies for SDN networks is to specify traffic forwarding through policies, where each policy rule identifies a traffic flow and its traversed service chains.

While network operators need to check network configurations as soon as possible, the SDN verification literature focuses on checking policy correctness during or after their deployment. This paper, instead, proposes early verification of forwarding policies before their deployment, by looking for the presence of anomalies that can potentially lead to erroneous and unexpected network behaviour. The proposed verification relies on a formal model that enables high flexibility in specifying both a forwarding policy and the set of anomalies to verify. The presented approach is efficient and highly scalable, as confirmed by tests with large networks.

Keywords: Service Function Chaining, Forwarding policies, Formal verification.

1. Introduction

A recent innovation in networking is the Service Function Chaining (SFC) concept [1], which consists in instantiating an ordered sequence of network functions, and consequently steering a particular portion of packets (e.g. the ones of a particular user) through the deployed chain.

However, SFC services have introduced additional complexity and many challenges in flow management, addressed with the introduction of Software Defined Networking (SDN) [2], which centralises the network management logic into a single programmable Controller.

Network operators can simply use one of the existing network programming languages to program the SDN controller and dictate the forwarding behaviour of the network at run-time. Those languages (e.g., Flow-based Management Language (FML), Frenetic and Merlin) provide different levels of abstraction for expressing network-wide policies.

This paper focuses on the concept of *forwarding policy*, which is a collection of rules for defining how packets must be classified and forwarded in a SFC.

After the specification phase, a forwarding policy is translated into the most suitable configuration of SDN switches, e.g. FlowTable entries for an OpenFlow switch[1], which the Controller will install to instantiate the desired service chains.

Network operators should check the presence of errors and ambiguities in policy specifications, otherwise faults in network configurations may arise at run-time. For example, an ambiguous specification may generate a conflict among flow entries in switches' FlowTables (e.g., two entries manage the same traffic flow, but one entry enforces the action “`drop`" while the other one enforces the action "flood" for the same flow).[2]

*Corresponding author

Email addresses: `fulvio.valenza@polito.it` (Fulvio Valenza), `serena.spinoso@polito.it` (Serena Spinoso), `riccardo.sisto@polito.it` (Riccardo Sisto)

[1]https://www.opennetworking.org/

[2]OpenFlow solves these conflicts by adopting priorities among

Even if some of the aforementioned programming languages perform their own validity checks before translating policy specifications into OpenFlow entries, these checks depend on the language adopted to program the network. Instead, in order to achieve best network reliability and security, the administrator should have a uniform checking mechanism, independent of the adopted controller language.

In addition, so far the literature has proposed many OpenFlow-oriented verification tools (e.g., [3, 4, 5, 6]) to check the violation of network invariants in the output of the forwarding policy translation (i.e., in the OpenFlow switches configurations). However, these tools detect problems during or after the switch configurations deployment. Instead, an *earlier detection*, done during the policy specification phase, would have two advantages. The first one is that, in case of error detection, error fixing is faster, because the fixing phase can start earlier, without even having to start the deployment phase. The second one is that, in case of error, the computational and storage resources necessary for translating the anomalous policy rules and for deployment are not wasted as otherwise happens.

An early verification is fundamental, especially in the new smart and IoT environments managed through SDN, that are becoming essential elements in the industrial network systems (INSs) with the advent of Industry 4.0 and Factory of the Future paradigms [7]. In these systems, where security and safety are strictly interdependent and productivity is one of the main goals, the introduction of mechanisms for detection of unexpected behaviours before the deployment phase would improve the security and safety of the systems. Moreover, it can also avoid the waste of computational resources from the side of the controller, due to translation and storage of erroneous inputs.

flow entries. However, the highest-priority entry may not be the most suitable one for managing that particular traffic flow, which can lead to anomalous behaviour.

Table 1: List of acronyms used in this paper.

DPI	Deep Packet Inspection
FML	Flow-based Management Language
FOL	First Order Logic
HSA	Header Space Analysis
IDS	Intrusion Detection System
PGA	Policy Graph Abstraction
SAT	Satisfiability Modulo Theory
SDN	Software Defined Networking
SFC	Service Function Chain
VPN	Virtual Private Network

In this paper, we mainly aim at enabling early error detection on forwarding policies, relying on a formal modelling approach. In practise, a precise and unambiguous meaning is given to a forwarding policy specification, independently of the adopted programming language and of its level of abstraction. A Forwarding policy will thus be expressed by means of a single formalism that embraces the variety of abstractions offered by the existing SDN programming languages.

By *anomaly*, we mean any erroneous or unwanted policy specification (e.g. including errors, conflicts or sub-optimizations), which may be due to e.g. human errors, and that may cause misleading network conditions and states. An example of anomaly is the violation of an operator-defined constraint of the SFC (e.g. network function ordering) or a conflict in the forwarding specifications. We assume the correctness of the translation algorithm that generates OpenFlow rules from policy rules, because it is generally implemented as an automatic process and thus we leave its verification out of scope.

Even if we eliminate anomalous policy specifications, other errors in network forwarding may still be present at run-time, due to wrong configurations installed into the network functions (e.g., wrong filtering rules installed in firewalls). Errors of this kind can be detected and solved by means of other approaches, such as the ones proposed in [8, 9, 10], that use complex network models and require more time-consuming verification algorithms. For this reason, the approach we are proposing does not substitute other more complex and accurate analyses, but it aims at early and fast detection of a number of anomalies already in the policy specification phase.

Another contribution of our approach is the possibility to define and verify custom anomalies specified by the operator, in addition to a set of pre-defined anomalies corresponding to general mistakes to be avoided in any network. This high flexibility in defining the anomalies to check is desirable because it enables the customisation of the verification process.

In our view, both custom and pre-defined anomalies can be specified using the same formalism, thanks to a set of operators that let one precisely and unambiguously specify the meaning of each anomaly. The anomalies specified by means of these operators are automatically translated into formulas in First Order Logic (FOL) that are finally fed to the verifier along with a policy to be checked. In this way, the user is not exposed to the complexity of FOL.

In this paper, we also propose a possible pre-defined set of anomalies to be detected. Such set includes novel anomaly classes proper of the SFC domain in addition to those classes of anomalies that lead to errors in the derived OpenFlow configurations and that have already been studied in the literature [11].

The remainder of this paper is organized as follows: Section 2 presents the current state of the art; Section 3 summarizes the problem statement and contributions of this work; Sections 4, 5, 6 respectively describe the structure of a forwarding policy, the supported operators for specifying anomalies and the anomaly detection model. We have also implemented an anomaly detection process, in order to evaluate the time required to verify policies for a whole network (Section 7). Finally, Section 8 concludes the paper and presents some possible future works.

2. Background

The most relevant works related to our approach can be divided into three categories, i.e. SDN verification, SDN programming languages and network policy analysis.

SDN verification. The current literature on SDN verification generally adopts two approaches: either off-line or real-time verification. The off-line OpenFlow-oriented verification tools take a snapshot of the global network behaviour, e.g. by collecting the forwarding entries installed into the network switches, or they model the behavior of the network when the SDN controller runs a particular SDN program, and check whether some basic invariants hold. A first example of off-line tool is NICE [12], which checks the presence of network invariants by combining model checking and symbolic execution approaches. Similarly, Anteater [13] verifies such invariants by expressing them as boolean satisfiability (SAT) problem instances while NetPlumber [4] relies on Header Space Analysis (HSA) in order to detect forwarding loops and leakage problems.

The real-time approach consists of placing the verification tool as a layer between SDN Controller and network switches. This is the case of VeriFlow [3], which dynamically checks if the absence of forwarding loops and black holes is satisfied at each OpenFlow rule insertion.

The main limitation of such off-line and real-time tools compared to what we are proposing here is that they do not perform an early detection of errors and faults.

SDN Programming languages. The literature presents a variety of SDN programming languages. Even though they do not focus on our main aim of checking network correctness, they share with our work the need to specify a forwarding policy and they also provide some form of checking on the policy that can be specified. For this reason, we analysed their variety in modelling packet forwarding in order to define a verification model flexible enough to fulfil the network operators' needs, while the checks they provide on policies can be considered as a basis for defining a set of pre-defined policy anomalies.

Frenetic [14] and NetCore [15] are two first examples. They let the user identify traffic flows by means of a low-level abstraction, i.e. predicates over standard OpenFlow headers (e.g. IP address, VLAN id, etc.) and operators like union and intersection applied on those predicates. The same level of abstraction is offered by Pyretic [16], which allows sequential and parallel policy compositions in addition to what is offered by Frenetic and NetCore. A similar approach is also adopted by Merlin [17], also which enables network operators to define forwarding policies with bandwidth constraints.

All such languages offer the possibility to perform some static checks on policy descriptions, but they all miss an underlying formal model of policy and anomaly, and each one of them performs only some specific and fixed checks over a forwarding policy (e.g. Merlin checks only if a policy modification introduced by tenants includes the chains enforced in the original policy set down by the operators; FML first detects a fixed set of conflicts and then it fixes them by exploiting resolution techniques; FatTire generates network configurations that are conflict-free by construction but does not provide other forms of checking). Hence, the use of a verification mechanism specific for the adopted language limits the set of errors and faults that can be detected.

Some of these languages also cover additional problems in network operation, like fault-tolerance for FatTire and bandwidth allocation for Merlin. On the contrary, we do not claim to cover such range of problems, but we only aim at detecting anomalies in forwarding policy specifications.

Network Policy analysis. The literature focuses mostly on detecting redundancies and conflicts among rules that make up a policy in several domains. Among such domains, the most relevant ones for our work are the filtering and the OpenFlow ones. In such domains, a conflict is generally seen as a faulty network state derived by two configuration rules (e.g. firewall rules and switch flow entries) that overlap and have different, conflicting actions. Redundancy of policy rules, instead, is generally considered a kind of sub-optimization in policy specification.

Part of our model has been inspired by the previous works on conflict analysis over such domains (e.g. [18], [19], [11]), but we have reinterpreted and extended them to be applied to the SFC domain, not only for conflicts or redundancy, and to be used also with a high-level modelling formalism.

In the filtering domain, Al-Sharer *et al.* [18] have proposed a tree-based representation of firewall configurations to check anomalies among filtering policies. In particular, the underlying formal model is able to detect an anomaly between two rules by checking which relationship exists between them. A similar approach has been proposed by Cuppens *et al.* [20], who have included the analysis of NID configurations. The main limitation of these works is that checking only relationships among rules limits the set of anomalies that can be detected, while we envision a model flexible enough to enable operators to define their own anomalies, in addition to a pre-defined set of anomalies to be checked in every network.

Other solutions have a similar limitation, like Liu *et al.* which focuses on building anomaly- and redundancy-free firewall configurations [21]. Such solutions check the relationships between rules two by two, while we envision a complete analysis over the whole set of rules that can detect anomalies triggered by one, two or many rules.

Regarding the OpenFlow domain, in addition to the fact that such proposals (e.g. [11], [22]) can perform a late SDN network analysis, most of them search for conflicts in OpenFlow configurations only, and overlook other kinds of misconfiguration (e.g. network function ordering). In this direction, an interesting and promising work was proposed by Prakash et al. [23]. Through a Policy Graph Abstraction (PGA) to express OpenFlow policies and an algorithm to automatically compose such policy graphs, the authors are able to determine an appropriate service order and to resolve policy conflicts, by minimizing operator interventions.

3. Problem Statement and proposed solution

As claimed above, all the aforementioned policy-oriented programming languages offer only some of static checks, and they miss an underlying formal model for both policy and anomaly specification. This means that an administrator can check only a limited and fixed set of errors and faults over the chosen forwarding policy rules, in case one of the existing SDN programming languages is used. Moreover, these checks are not based on mathematically rigorous models.

Our aim is, thus, to offer administrators, on one side, a single mechanism that can check a richer set of anomalies in their forwarding policy rules and, on the other side, a mechanism to define their own anomalies to check. A mathematical foundation is given to these mechanisms by defining a formal language to specify a forwarding policy and a FOL-based model to specify custom anomalies and to detect their presence in the policy rules that will be enforced in the network, can be performed.

In particular, our formal language and model rely on a middle-level of abstraction between the high-level representation adopted by the existing SDN programming languages and the low-level rules installed into the network switches (e.g. OpenFlow rules). In this way, both top-down and bottom-up anomaly analysis.

In the former, administrators specify the preferred forwarding policy by using one of the SDN programming languages; such policy will be translated into our formalism to be checked by our anomaly model. In case no anomalies are identified, policy rules can be translated into the low-level configurations that will be installed into the network.

The bottom-up analysis, instead, may be applied in case an administrator changes manually the network configuration (e.g. by adding a new OpenFlow rule in a switch FlowTable), willing to make sure of the new network configuration correctness. Thus, starting from the low-level rules installed in the network nodes, these are mapped into our policy model which is analysed for checking the presence of anomalies. The literature has proposed many OpenFlow-oriented verification tools that can detect the presence of anomalies introduced by switches configurations updates. However, these tools can follow only the bottom-up approach and cannot perform a top-down analysis.

In our view, an anomaly represents any erroneous and undesired condition that an administrator wants to detect and eliminate in a forwarding policy, in order to guarantee a self-consistent policy and avoid some traffic forwarding errors in the network at run-time. We are targeting not only conflicts among policy rules but also, for example, anomalies triggered by a single rule, such as the violation of an ordering constraint in the sequence of functions specified by a single rule. For example, a network operator wants to ensure that a NAT is always configured to process traffic before a firewall. This means that we have to detect the anomalous situation when a NAT is located after a firewall in the SFC topology. Another example is when an administrator wants to speed up web services response by making sure all web traffic between users and their servers traverses a web cache.

In summary, we propose a verification approach that is: *(i)* independent from the overhead language adopted to program the network; *(ii)* flexible enough to cover a large set of anomalies (i.e. non only conflicts); *(iii)* general enough to enable the use of the different levels of abstraction allowed by the aforementioned languages. Our model, in fact, can be integrated into any SDN Controller programmable with a policy-oriented language. The only thing network operators need is the addition of a language-specific module to map the forwarding policy from the adopted language into our model or vice-versa. However in this paper we are mainly interested in presenting the details of the model and its features, while the design of translation modules is left as future work. In the next sections, we present our formal model for both policy and anomalies in more detail, and we provide also a rich set of anomaly examples.

4. Forwarding policy model

In our model, a forwarding policy ($\mathbb{R}_\mathbb{F}$) is a set of *forwarding rules* (or simply *rules*), each one putting in relation traffic flows with the SFCs those flows can traverse at run-time. A generic forwarding rule r in a forwarding policy ($r \in \mathbb{R}_\mathbb{F}$) has the following structure:

$$r = (\mathcal{M}, \mathcal{C}, \mathcal{P}), \; r \in \mathbb{R}_\mathbb{F} \tag{1}$$

where:

- $\mathcal{M}$ is the traffic flow managed by the rule, which belongs to the set of all possible flows in a network ($\mathcal{M} \in \mathbb{M}$);
- $\mathcal{C}$ is the set of SFCs that $\mathcal{M}$ can potentially traverse at run-time and it is a subset of the whole set of chains instantiated in the network ($\mathcal{C} \subseteq \mathbb{C}$);
- $\mathcal{P}$ is the set of *Properties* associated to the flow $\mathcal{M}$ and to the set of SFCs $\mathcal{C}$ that $\mathcal{M}$ can potentially traverse ($\mathcal{P} \subseteq \mathbb{P}$).

Note that in our model we suppose that not necessarily all the packets of a flow $\mathcal{M}$ traverse all the configured chains at run-time. In a real scenario, packet forwarding, in fact, depends also on network function configuration and state, thus a flow can be forwarded to zero, one, many or all of the allowed chains, and individual packets belonging to a flow can traverse different chains. As an example, let us consider a flow $\mathcal{M}$ defined as all web traffic with a given source address, and let us assume we want this flow to be allowed to reach only a collection of web servers, all behind a load balancer, which selects at runtime the destination

Table 2: List of notations and symbols used in this paper.

r	forwarding rule
a	anomaly
n_i^g	is the g-th network field in r_i
v_i^g	is it is the value associate to n_i^g
c_i^k	specifies the k-th chain in the i-th rule
$f_i^{w_k}$	is the w-th function in c_i^k
$\mathcal{M}$	traffic flow managed by the rule
$\mathcal{C}$	SFCs that $\mathcal{M}$ can potentially traverse at run-time
$\mathcal{P}$	properties associated to the flow $\mathcal{M}$
$\mathcal{N}$	network fields
$\mathcal{N}^H$	packet header fields
$\mathcal{N}^N$	high-level names
K	Knowledge base
$\mathbb{R}_\mathbb{F}$	forwarding policy
$\mathbb{M}$	set of all traffic flows
$\mathbb{C}$	set of all SFCs
$\mathbb{P}$	set of all Properties
$\mathbb{K}$	set of all Knowledge
$\mathbb{K}$	set of all Anomalies
$\mathcal{M}^N$	$\mathcal{M}$ in name-based representation
$\mathcal{M}^\langle$	$\mathcal{M}$ in header-based representation
usr_src	sender
usr_dst	receiver
f_type	traffic type
eth_src	ethernet source
eth_dest	ethernet dest
eth_src	ethernet source
eth_dest	ethernet dest
$vlan\_id$	vlan id
ip_src	ip source
ip_dest	ip dest
ip_proto	ip proto
$port\_src$	port source
$port\_dest$	port dest
Ψ	Translation function
$\pi(f)$	the position of the f in the chain
$=$	equivalence
$\succ$	dominance
$\succeq$	equivalence or dominance
$>$	majority
$\sim$	correlation
$\perp$	disjointness
$\subset$	inclusion
$\in$	membership
[]	ordered sequence
{}	set

web server for each packet, based on its internal algorithm and state. In this case, we can write a policy that associates $\mathcal{M}$ with a set of chains, each one including the load balancer and one of the destination web servers. In other cases, the packets of a flow can even traverse more than one chain at a time. This happens, for example, with a mirroring function that replicates the incoming flow onto different outgoing chains. Note also that the proposed model does not consider rule priorities as instead it has been done in the OpenFlow domain ([11], [22]). This is because we are working at a higher abstraction level, where we loose the notion of order among forwarding rules. It is only when a forwarding policy is translated into OpenFlow flow entries that we need a priority in FlowTables. Another reason for omitting priorities is also that each forwarding rule specifies *all* the allowed chains for a set of flows. For this reason, in order to avoid ambiguity in a policy, forwarding rules should be specified with non-overlapping traffic flows. When this condition is violated, we have an anomaly in the policy according to our model.

Since network operators should not be limited to use one particular SDN programming language (e.g. FML, Merlin, Pyretic) and each language has its own formalism and abstraction level, our model has been designed with two levels of abstraction for specifying traffic flows. Generally, a flow $\mathcal{M}$ is modelled by referring to a set of *network fields*. A network field n is an element of $\mathcal{N}$ ($n \in \mathcal{N}$) and the definition of $\mathcal{N}$ varies based on the level of abstraction we adopt. In particular to model $\mathcal{M}$, we rely either on a set $\mathcal{N}^H$ of packet header fields (i.e. *header-based representation*) or on a set $\mathcal{N}^N$ of high-level names (i.e. *name-based representation*). A *name* is a label defined by network operators to represent elements in their networks, such as hosts, network functions, traffic types, VLANs and subnets. A set of names can thus be used to identify a particular traffic flow, for example the one of an SSH connection from a particular user to a particular external subnet.

The verification workflow of our model consists of receiving the forwarding policy (expressed in one of the two formalisms), performing the verification step, and reporting the anomalies detected, otherwise continuing with the SFC deployment. An additional step is performed in case policy rules are expressed in the name-based abstraction. In this case, after the first verification step, the high-level policy is translated into the corresponding header-based representation and its correctness is checked again. This two-fold check enables high flexibility in the formalism to adopt, along with a more complete and early anomaly detection.

In the next sub-sections we present the details of how traffic flows are modelled and how the name-based representation can be mapped onto the header-based one.

4.1. Name-based representation

In the name-based representation, $\mathcal{N}^N$ can be defined as follows:

$$\mathcal{N}^N = \{usr\_src, usr\_dst, trf\_type\} \tag{2}$$

where each field n^n of this set has a defined meaning. For example, in (2), the fields indicate respectively the sender, receiver and traffic type that characterize a traffic flow, but this set can be extended as needed to include more fields.

Generally, each network field n^n has a type, i.e. the set of values that can be taken by the field. A value v has to be specified for each field n^n of $\mathcal{N}$ in order to identify a traffic flow. In addition to specifying single values, it is also possible to specify sets of values or even any value, which is represented by the special symbol *. Hence, a flow $\mathcal{M}^N$ in the name-based representation ($\mathcal{M}^N \in \mathbb{M}^N$) is formally defined by a set of equalities, one for each n^n:

$$\mathcal{M}^N = (usr\_src = v_{usr\_src}, usr\_dst = v_{usr\_dst},$$
$$trf\_type = v_{trf\_type})$$

Examples of values for trf_type are single values like "`tcp`", "`udp`", "`ftp`", "`ssh`" or sets of names like "{`http`, `https}"`[3], while usr_src and usr_dst can indicate for example users, hosts, subnets, VLANs, etc... (e.g., "*User1*", "*Department1*" and "*Turin*").

4.2. Header-based representation

We also provide another way to model a traffic flow $\mathcal{M}$, where network fields refer to standard packet header fields. In particular, our header-based representation of a flow $\mathcal{M}^H$ currently relies on a set $\mathcal{N}^H$ of OpenFlow fields n^h, but of course this set can be extended too. More precisely, $\mathcal{N}^H$ is currently defined as follows:

$$\mathcal{N}^H = \{eth\_src, eth\_dst, eth\_type, vlan\_id, \\ ip\_src, ip\_dst, ip\_proto, port\_src, port\_dst\}$$

Similarly to the $\mathcal{M}^N$ model, each n^h is characterised by a type and, in order to specify a flow, it must be assigned either a single value or a set of values or all values (*). A set of values can be expressed as a range in case the type is a totally ordered set. For example, in a flow specification we can use $ip\_dst = 8.8.8.0/24$ or $port\_dst = [80, 100]$, because the types of IP address and port number fields are totally ordered sets of values, but if we prefer we can also use single values (e.g., $ip\_dst = 8.8.8.151$ or $port\_dst = 80$).

Hence, a flow $\mathcal{M}^H \in \mathbb{M}^H$ is formally defined as follows:

$$\begin{aligned}\mathcal{M}^H = (&eth\_src = v_{eth\_src}, eth\_dst = v_{eth\_dst}, \\ &eth\_type = v_{eth\_type}, vlan\_id = v_{vlan\_id}, \\ &ip\_src = v_{ip\_src}, ip\_dst = v_{ip\_dst}, \\ &ip\_proto = v_{ip\_proto}, port\_src = v_{port\_src}, \\ &port\_dst = v_{port\_dst})\end{aligned}$$

Our model supports the translation of the name-based representation into the header-based one. We suppose this process is performed by an additional entity, named *policy engine*, similar to the one provided at runtime by programming languages based on names (e.g. FML).

Our policy engine uses a knowledge base $\mathcal{K} \subseteq \mathbb{K}$ that is a set of mappings from high-level names to corresponding low-level values. Formally, $\mathcal{K}$ is a set of entries k, each one being a set of name-value pairs, where the value paired with "name" is the high-level name mapped by the entry and the other values are the corresponding low-level values. The entries $k \in \mathcal{K}$ can include different low-level values according to the type of high-level name they map.

The algorithm used by the policy engine to map a name-based flow specification $\mathcal{M}^N \in \mathbb{M}^N$ into its corresponding header-based $\mathcal{M}^H \in \mathbb{M}^H$ using a knowledge base $\mathcal{K} \subseteq \mathbb{K}$ is formally represented by a function Ψ:

$$\Psi : \mathbb{K} \times \mathbb{M}^N \mapsto \mathbb{M}^H, \quad \Psi(\mathcal{K}, \mathcal{M}^N) = \mathcal{M}^H$$

This function translates each field of $\mathcal{M}^N$ into one or more fields of $\mathcal{M}^H$. Note that the header-based fields into which each name-based field is translated can depend on the knowledge base. In our specific setting, they depend on the types (client or server) of the end users involved.

4.3. SFC representation

As specified in (1), a forwarding rule also includes the *service chains* (i.e., SFCs) C that can be traversed by the flow $\mathcal{M}$. In detail, C is the set of chains c enforced by a rule, which is, in turn, a sub-set of all possible chains $\mathbb{C}$:

$$C = \{c^1, c^2 ..., c^n\}, \quad c^k \in C \subseteq \mathbb{C}$$

Each chain $c^k \in C$ is represented in our model as an ordered sequence of network functions $c^k = [f^{1_k}, f^{2_k}, ..., f^{m_k}]$. Each function f^{w_k} in a chain c^k is one of the functions present in the network and it is modelled by the pair:

$$f^{w_k} =< f\_id^{w_k}, f\_type^{w_k} >$$

where $f\_id^{w_k}$ is the function identifier and $f\_type^{w_k}$ is the function type, which necessarily has to belong to the catalogue of network functions ($\mathbb{F}$) offered by the opera-

[3]trf_type can be initialized with any other name of well-known protocols.

tor[4]. Thus we model a network chain as:

$$c^k = [< f\_id^{1_k}, f\_type^{1_k} >, ..., < f\_id^{m_k}, f\_type^{m_k} >],$$
$$f\_type^{w_k} \in \mathbb{F}$$

This approach offers a level of detail in modelling SFCs higher than the one offered by existing formalisms. The SDN programming languages that explicitly manage service chains (e.g. Merlin, FatTire) generally indicate just the types of functions, without being able to consider their real instances deployed into the network. Thanks to our approach, instead, network operators can describe their networks more precisely and perform more accurate checks. To summarise, given a forwarding rule r, its name-based representation is:

$$r = (\mathcal{M}^N, C) = ((usr\_src = v_{usr\_src}, usr\_dst = v_{usr\_dst}, trf\_type = v_{trf\_type}), \{[< f\_id^{1_1}, f\_type^{1_1} >, ...], ...\})$$

while its header-based representation is:

$$r = (\mathcal{M}^H, C) = ((eth\_src = v_{eth\_src}, eth\_dst = v_{eth\_dst}, eth\_type = v_{eth\_type}, vlan\_id = v_{vlan\_id}, ip\_src = v_{ip\_src}, ip\_dst = v_{ip\_dst}, ip\_proto = v_{ip\_proto}, port\_src = v_{port\_src}, port\_dst = v_{port\_dst}), \{[< f\_id^{1_1}, f\_type^{1_1} >, ...], ...\})$$

4.4. Properties representation

A properties set $\mathcal{P}$ is modelled by referring to a set of *Properties*. In our model, the properties enable network administrators to specify several requirements useful to mange system resource and/or quality of service (QoS). In particular, a property p ($p \in \mathbb{P}$) is a kind of resource allocated to a specific traffic flow $\mathcal{M}$. Possible examples of properties, such us Bandwidth and VNF node computational power, have been listed in [24, 25].

$\mathcal{P}$ is thus formally defined by a set of equalities between a property p and its numerical value v.

$$\mathcal{P} = (p_1 = v_{p_1},\ p_2 = v_{p_2},\ ...,\ p_n = v_{p_n})$$

We considered the value v_p as a minimum value. This means that for the traffic flow $\mathcal{M}$ and for the C at least the value v_p has to be allocated for the system resource indicated by p.

It is also possible to specify any value for a property, by means of the symbol $*$ (i.e., no requirements about minimum quantity of resources are specified).

The next sections first introduce the relational operators that can be used for building anomaly specifications. Such operators enable pairwise comparisons between the elements that compose a forwarding rule or that belong to different rules. We then introduce the anomaly model, which is a FOL formula that involves a set of pairwise comparisons.

To help the readers, from now on, in this paper we indicate the elements of a forwarding rule r_i as follows:

- $\mathcal{M}$ is the traffic flow specification, regardless if its being name-based or header-based (i.e. $\mathcal{M}^N$ and $\mathcal{M}^{\mathcal{H}}$);
- $\mathcal{M}_i$, C_i and $\mathcal{P}_i$ are respectively the flow, the SFCs, and the properties dictated by rule r_i;
- n_i is a generic network field of rule r_i;
- n_i^g is the g-th network field in r_i and v_i^g is its value;
- c_i^k specifies the k-th chain in the i-th rule;
- $f_i^{w_k}$ is the w-th function in c_i^k.

We use this notation in case we are referring to different forwarding rules (e.g., r_i and r_j), while in case we are indicating a single rule, we do not use any index as subscript to indicate the rule itself r and its elements.

[4]In this paper, we use abbreviations in the formulas to indicate the type of network function involved in the network chains. In particular, we use these abbreviations in the next examples: HOST (End Host), SERVER (Web Server), FW (firewall), NAT (network address translator), DPI (deep packet inspection), MN (monitor), LB (load-balancer), SPAM (anti-spam), CACHE (web-cache), IDS (intrusion detection system), VPN (virtual private network) and L7_FW (layer 7 firewall).

5. Relational operators for anomaly specification

In order to enable the specification of anomalies, the model offers a set of relational operators. These operators enable the specification of pairwise comparisons ($x \in \mathcal{X}$), each one involving network fields, SFCs and proprieties belonging to the same or to different rules. Formally, these comparisons are predicates that let us finally identify sets of matching forwarding rules. More precisely, if x is a comparison that involves fields and SFCs belonging to the same generic rule r, x can be regarded as a function of r which returns the result (true or false) of the comparison evaluated on r. Moreover, x identifies the set of rules r such that $x(r)$ is true. If instead x involves fields and SFCs belonging to two different rules r_i, r_j, then x can be regarded as a function of two variables r_i, r_j which returns the result (true or false) of the comparison evaluated on r_i and r_j. Moreover, x in this case identifies the set of pairs of rules (r_i, r_j) such that $x(r_i, r_j)$ is true.

Note that for simplicity, in the following subsection, we do not describe the operators used to evaluate properties. This is because we use the same operators used for numeric values (i.e., $=, \neq, >, <$).

5.1. Network field operators

We have defined a set of relational operators to compare network fields in the name-based representation and in the header-based one. Even though the value-type of a network field may depend on which one of the two representations is used, the meaning of our relational operators remains unchanged.

In order to perform a pairwise comparison between network fields and, in turn, to check when one or more forwarding rules match an anomaly, we need to establish inclusion relationships among network fields. The inclusion relationships involving sets of values and single values derive naturally from the set inclusion concept. For example, in the header-based representation, a range of network addresses includes single IP addresses (similarly for a port range and single port numbers).

However, in the name-based abstraction, where symbolic names are used, we may need to extend the inclusion relationship by also considering the meaning of symbolic names. For example, for the trf_type field, which specifies one or more network protocols, we suppose to use the native inclusion relation among the network protocols over the different levels of the ISO/OSI stack. In particular, a network protocol of a stack layer includes a set of protocols in the above layer and it is also included by a protocol of the underlying layer. For example, the Layer_4 TCP protocol includes many Layer_7 protocols (e.g., HTTP, FTP) and it is included by the Layer_3 protocol IP. For what concerns instead usr_src and usr_dst, we suppose that a user name can be associated to a host name, which in turn belongs to a subnet: for example, the user "*Alice*" is associated to the host name "*HostA*" that is included in the sub-net "*Department1*". These relations that bind names have to be specified by the operator and are added to the knowledge base of the policy engine, so that the verifier can take them into account.

The supported operators are listed below.

- **equivalence (=)**: two fields are equivalent if the value(s) they can take (in the flows of the rules they belong to) are the same, even though they are different fields. For example, if $\mathcal{M}^{\mathcal{H}}$ includes $port\_src = 80$ and $port\_dst = 80$, then, for rule r, which includes $\mathcal{M}^{\mathcal{H}}$, we have $port\_src = port\_dst$. While, if $\mathcal{M}^{\mathcal{N}}$ has $usr\_src = Alice$ and $usr\_dst = Alice$, similarly to the previous case, $usr\_src = usr\_dst$. Of course, equivalence can also be applied to fields belonging to the flows of different rules;

- **dominance (>)**: a field dominates another one if it can take all the values that can be taken by the other field. For example, if $port\_src_i = [1024, 2048]$ and $port\_src_j = [1024, 1500]$, then $port\_src_i > port\_src_j$.

Another example is if $trf\_type_i = \{http, https\}$ and $trf\_type_j = \{http\}$, then $trf\_type_i > trf\_type_j$. Inclusion relations over names are also taken into account. For example, let us suppose $trf\_type_k = \{tcp\}$ and $trf\_type_j = \{http\}$, then $trf\_type_k > trf\_type_j$ thanks to the above mentioned inclusion relation over protocols. Generally, this operator makes sense for fields that can take sets of values rather than single values;

- **majority** ($>$): this operator can be applied only to fields that are specified to take single numeric values. In this case, a network field is greater than another one if their values have this relation (e.g., if $port\_src = 70001$ and $port\_dst = 65535$ then $port\_src > port\_dst$);
- **correlation** ($\sim$): two fields are correlated if they share some values but none dominates the other. Generally, this operator makes sense for fields that can take sets of values rather than single values. For example, if $port\_src_i = [1024, 1500]$ and $port\_src_j = [0, 1100]$, then $port\_src_i \sim port\_src_j$. Similarly, if $trf\_type_i = \{pop3, imap, snmp\}$ and $trf\_type_j = \{ftp, snmp\}$, then $trf\_type_i \sim trf\_type_j$;
- **disjointness** ($\perp$): two network fields are disjoint if they do not share any value. For instance, if $port\_src_i = [1000, 1024]$ and $port\_src_j = [1100, 8080]$, then $port\_src_i \perp port\_src_j$. Similarly, if $trf\_type_i = \{http, https\}$ and $trf\_type_j = \{imap, imaps\}$, then $trd\_type_i \perp trf\_type_j$.

The model also offers the negative form of the aforementioned operators, like *non-disjointness* ($\not\perp$ - two fields are either equal, correlated or one dominates the other) and *non-equivalence* ($\neq$ - two fields can be correlated, disjoint or one can dominate the other).

Also, the above operators can be applied not only to compare two fields but also fields and single values or fields and sets of values with the obvious meaning.

Moreover, combinations of operators are allowed. This is the case of *equivalence or dominance* ($\geq$ - i.e., a network field is equivalent to or dominates another one), *equivalence or correlation* ($\simeq$ - i.e., a network field is equivalent or correlated to another one) and *equivalence or majority* ($\geq$ - i.e., a network field is equivalent to or greater than another field).

5.2. SFC operators

As already mentioned, here we define new operators, in order to enable comparisons that involve SFCs. First, we introduce the following notation to represent ordered sequences (i.e., SFCs) and unordered sets of network functions:

- **ordered sequence** ([]): this notation was already introduced for the specification of SFCs inside forwarding rules. It is also used to represent ordered sequences of network functions in an anomaly specification. Via the wild card character *, the proposed model supports the specification of unidentified functions, i.e. functions for which only the type is specified, not the identity. For example, a chain composed of a NAT followed by a firewall can be specified generically as $[< *, NAT >, < *, FW >]$;
- **set** ({}): this notation can be used to specify unordered collections of functions. For example, a chain including an application firewall ($L7\_FW$) and a DPI, not necessarily in this order, can be specified as $\{< *, L7\_FW >, < *, DPI >\}$.

For what concerns the comparison between SFCs that can belong to the same forwarding rule or to different rules, we extend the current literature by enabling pairwise comparisons between: *(i)* two chains of either the same or different rules; *(ii)* a chain and an ordered sequence of functions (i.e., a chain not managed by a forwarding rule); *(iii)* a chain and a set of functions. In some cases, the same operators can be used for different types of comparisons, the exact meaning of the comparison being determined by the types of the compared elements.

In case of comparison between two chains (of the same or of different rules - e.g., c^k and c^l) or a chain and an ordered sequence (e.g., c^k and $[f^1, f^2, ..]$), the following operators can be used:

- **equivalence** $(=)$: two chains are equivalent if they are the same ordered sequence of network functions (e.g., if $c^k = [f^1, f^2]$ and $c^l = [f^1, f^2]$, then $c^k = c^l$);
- **dominance** $(\succ)$: a chain dominates another one when it contains the second chain as a subsequence and the two chains are not equivalent (e.g., if $c^k = [f^1, f^2, f^3]$ and $c^l = [f^2, f^3]$, then $c^k \succ c^l$);
- **correlation** $(\sim)$: two chains are correlated if none dominates the other, but they share an ordered sub-chain (e.g., if $c^k = [f^1, f^2, f^3]$ and $c^l = [f^4, f^2, f^3]$, then $c^k \sim c^l$);
- **disjointness** $(\perp)$: two chains are disjoint if they do not have any sub-chain in common (e.g., if $c^k = [f^1, f^2, f^3]$ and $c^l = [f^4]$, then $c^k \perp c^l$).

The comparison between a chain and an unordered set of functions (i.e., c^k and $\{f^1, f^2, ...\}$), instead, can involve the following operators:

- **correlation** $(\sim)$: a chain is correlated to a set of functions if it contains some of those functions (e.g., $c^k = [f^1, f^2, f^3] \sim \{f^4, f^2, f^3\}$);
- **disjointness** $(\perp)$: a chain and a set of functions are disjoint if they do not share any function (e.g., $c^k = [< spam, SPAM >, < fw, FW >, < dpi, DPI >] \perp \{< mn, MN >\}$);
- **inclusion** $(\subset)$: a set of non-ordered functions is included into a chain if all of its functions are part of the chain (e.g., if $c^k = [< spam, SPAM >, < nat, NAT >, < fw, FW >]$, then $\{< nat, NAT >\} \subset c^k$).

It is interesting to note that in some particular cases, the inclusion and dominance operators take the same meaning. Let us consider, for example, that one wants to specify the condition that a network function f belongs to a chain c. This condition can be expressed either by the comparison $\{f\} \subset c$ or by $c \succ [f]$. However, it is better to have different operators, in order to cover a richer set of anomalies and keep expressions as simple as possible. For example, if the model would support only the dominance operator, a network operator could specify that m functions f^1,f^2,..., f^m, not necessarily in this order, belong to a SFC, by specifying: $c \succ [f^1] \wedge c \succ [f^2] \wedge .... \wedge c \succ [f^m]$ By also supporting the inclusion operator $(\subset)$, we then make the formula syntax less complex and less likely to be mistaken: $\{f^1, f^2, ..., f^m\} \subset c$

For SFC comparisons too, the model offers the negative forms of the aforementioned operators (e.g., *non-correlation* $(\nsim)$, *non-dominance* $(\nsucc)$, etc.), and some combinations of operators (i.e., *equivalence or dominance* $(\succeq)$, *equivalence or correlation* $(\simeq)$ and *inclusion or equivalence* $(\subseteq)$). The support of negative and combined operators enriches the expressiveness of our model in describing the anomalies to check. Even though network operators could exploit existing formalisms like Merlin and FatTire for specifying forwarding policies, they will miss the high-level flexibility offered by our model in defining the anomalies to check. A feature we offer is the possibility to define both a positive and a negative form of an anomaly (e.g., "`Traffic from User1 passes through FW`" or "`does not pass through FW`").

In order to further enlarge expressivity, we define also another new operator that lets us specify comparisons related to the position of a function within a service chain:

- $\pi(f, c)$ returns the position of function f within chain c, if $\{f\} \subseteq c$, or 0 otherwise. Let us consider for example $c = [< nat, NAT >, < fw, FW >]$. The NAT position inside c is $\pi(< nat, NAT >, c) = 1$.

Finally, the model also allows one to check the membership of a chain c^k within a set of chains C_i:

- **membership** ($\in$): this boolean operator returns true if a chain c^k belongs to the set of chains C_i of the forwarding rule r_i and false otherwise. For example, let us consider $c^k = [< spam, SPAM >, < fw, FW >, < dpi, DPI >]$ and $C = \{[< nat, NAT >, < fw, FW >], [< spam, SPAM >, < fw, FW >, < dpi, DPI >]\}$, then $c \in C$ is true. The model supports also the negative form of this operator (i.e, $\notin$).

6. Anomaly model

Formally, an anomaly $a \in \mathbb{A}$ is a predicate defined on one or more rules, where $\mathbb{A}$ is the whole set of forwarding anomalies (or simply anomalies) defined by a network operator.

For example, if r is a variable that represents a rule, an anomaly can be formally represented by a function $a(r)$ that returns the boolean true if the anomaly is present in the single rule r and false otherwise. Similarly, if r_i and r_j are two rules, an anomaly can be defined as a function $a(r_i, r_j)$ that returns the boolean true if the anomaly is present in the pair of rules (r_i, r_j). More generally, an anomaly that involves n rules $r_1, r_2, ..., r_n$, can be represented by a function $a(r_1, r_2, ..., r_n)$ that returns the boolean true if the anomaly is present in this set of rules and false otherwise. A policy is anomaly-free if :
$\forall a \in \mathbb{A}\ a(r_1, ..., r_n) = false\ \forall (r_1, \ldots, r_n) \in \mathbb{R}_{\mathbb{F}} \times \cdots \times \mathbb{R}_{\mathbb{F}} | n \geq 1$ according to the arity of a.

In detail, an anomaly is formally specified by a set of Horn clauses that involve pairwise comparisons. Each clause is a conjunction of positive comparisons x_i on rule fields and chains, which implies the presence of the anomaly in a single rule or in a pair of rules. Hence, the structure of Horn clauses that define anomalies is as follows:

$$\begin{aligned} x_1 \wedge x_2 \wedge \ldots \wedge x_q &\rightarrow a(r), \quad a \in \mathbb{A} \\ x_1 \wedge x_2 \wedge \ldots \wedge x_q &\rightarrow a(r_1, ..., r_n), \quad a \in \mathbb{A} \end{aligned} \tag{3}$$

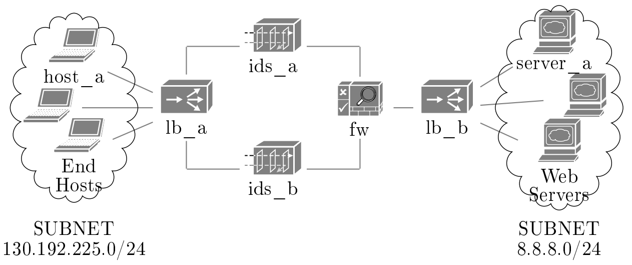


Figure 1: Topology example.

In practice, the intersection of the sets of rules identified by the comparisons that occur in the left hand side of the formula is the set of rules in which the anomaly is present.

In order to be flexible enough, the model supports also existential ($\exists$) and universal ($\forall$) quantification over SFCs in the left hand side of the Horn clauses to specify that some comparisons have to be satisfied by at least one or by all the SFCs of a forwarding rule[5].

An example of anomaly that refers to pairs of rules and that uses universal quantification is the rule duplication anomaly, which occurs when a policy includes two identical rules. This anomaly can be specified as the anomaly that is true when the pairwise equivalence between all the elements of two rules expressed in the header-based representation (r_i and r_j), including all the SFCs, is satisfied:

$$\begin{aligned} & eth\_src_i = eth\_src_j \wedge eth\_dst_i = eth\_dst_j \wedge \\ & eth\_type_i = eth\_type_j \wedge vlan\_id_i = vlan\_id_j \wedge \\ & ip\_src_i = ip\_src_j \wedge ip\_dst_i = ip\_dst_j \wedge \\ & ip\_proto_i = ip\_proto_j \wedge port\_src_i = port\_src_j \wedge \\ & port\_dst_i = port\_dst_j \wedge c_i^k \in C_j, \forall c_i^k \in C_i \wedge \\ & c_j^l \in C_i, \forall c_j^l \in C_j \wedge ban_i = ban_j \wedge mem_i = mem_j \wedge \\ & ram_i = ram_j \wedge ncpu_i = ncpu_j \rightarrow Duplication(r_i, r_j) \end{aligned} \tag{4}$$

[5]In this model, when we quantify universally on pairs of chains, we are considering implicitly pairs of different chains. For example, in case we check the correlation among the SFCs in a forwarding rule, we can specify $c_i^k \sim c_i^l,\ \forall c_i^k, c_i^l \in C_i$ to check only pairs of different chains, without indicating explicitly that $k \neq l$.

Since we aim at checking the presence of anomalies among forwarding rules expressed both in the name-based and in the header-based abstraction, our model needs to express anomalies with reference to both abstraction levels. To better understand how anomalies are expressed in both formalisms, let us consider another example of anomaly with reference to Figure 1. A network operator, who wants to make sure all web traffic between user Alice and Google servers traverses a firewall, can define a custom anomaly triggered if such web traffic may not traverse a firewall. This anomaly, which involves a single rule, is expressed in the name-based representation as

$$usr\_src = Alice \wedge usr\_dst = Google \wedge trf\_type = http \wedge$$
$$\{<*, FW>\} \not\subset c^k, \forall c^k \in C \rightarrow webNoFirewall(r)$$

while in the header-based representation, if we assume Alice has IP address 130.192.225.116 and Google servers have IP 8.8.8.0/24, the same anomaly is represented as

$$eth\_src =^* \wedge eth\_dst =^* \wedge eth\_type = 0\mathbf{x}0800 \wedge vlan\_id =^*$$
$$\wedge\, ip\_src = 130.192.225.116 \wedge ip\_dst = 8.8.8.0/24 \wedge$$
$$ip\_proto = 0\mathbf{x}06 \wedge port\_src =^* \wedge port\_dst = 80 \wedge$$
$$\{<*, FW>\} \not\subset c^k, \forall c^k \in C \rightarrow webNoFirewall(r)$$

As anomaly specifications are FOL formulas with predicates defined over policy rules, they inherit the standard formal semantics of FOL formulas, while the semantics of predicates is determined by the semantics of the relational operators we have defined. The presence of anomalies can be detected in a policy specification by means of general purpose tools such as automated theorem provers or production rule systems, as it will be shown in Section 7. The correctness of verification results depends on the correctness of such tools, which is generally assumed.

6.1. Anomaly Types

In the this subsection, we identify a number of types of anomalies, giving examples for each of them. Some of these examples can be considered as part of the set of predefined anomalies. Note that the pre-defined anomalies that we present as examples are not exhaustive, since we do not claim to cover the whole set of anomalies that arise in the SFC domain. The main aim of this paper is to enable the formal specification of policy rules and anomalies and the detection of a consistent number of anomalies as early as possible. Note also that the proposed anomalies do not include those anomalies that are combinations of the presented types. Of course, the model is able to treat and detect also this kind of anomalies. However, we preferred to limit our presentation to these type of anomalies, and encourage network operators to build customized ones.

The types of anomalies that we are proposing are based mostly on the object of comparison rather than on the comparison operator occurring in the formula. For this reason, in the following formulas that we use for defining anomaly types, we leave the operator unspecified and we indicate it generically by the $\star$ symbol.

6.1.1. Single-Field anomalies

The anomalies in this class are those that involve only comparisons between single network fields and specific values. Thus, following the generic anomaly structure defined in (3), a comparison x that composes a single field anomaly is expressed as: $x = n \star v$

Where the symbol v represent any well-formed values that it may be compared with the network field.

Examples of anomalies of this class are the ones triggered when the sender user refers to a non authorized name (e.g. a name "u" that is in the list of unauthorized names "$FakeUsers$") or when port numbers are grater than their maximum values. Such kind of anomalies belong to the set of pre-defined anomalies, since they are mistaken policy specifications in every network topology. They can be expressed by the following formulas:

$$usr\_src = u | u \prec FakeUser \rightarrow BadUserSrc(r) \quad (5)$$

$$port\_src > 65535 \rightarrow BadPortSrc(r) \quad (6)$$

$$port\_dst > 65535 \rightarrow BadPortDst(r) \quad (7)$$

6.1.2. Field-Pair anomalies

A Field-Pair anomaly is one that contains only pairwise comparisons of different network fields belonging to the same forwarding rule, like source and destination IP. Thus a generic Field-Pair anomaly is one expressed by means of comparisons that take the following form:

$$x = n^o \ \star \ n^p$$

An example of Field-Pair anomaly is when source and destination users (or IP addresses) are the same, which is specified by means of the following formulas:

$$usr\_src = usr\_dst \rightarrow BadUserName(r) \tag{8}$$

$$ip\_src = ip\_dst \rightarrow BadIpAddress(r) \tag{9}$$

Of course, these are other examples included in the predefined set of anomalies supported by our model.

6.1.3. Node Traversal anomalies

These anomalies are those that arise when a traffic flow can (or cannot) traverse one or more network functions. In this model, we specify the traffic flow by comparing network fields with specific values (i.e., $x = n \ \star \ v$). Hence, such anomalies are expressed by comparing network fields with specific values and chains with ordered or non-ordered sets of functions. The forms of comparisons in these anomalies are:

$$x = n \star v, \quad x = c \ \star \ [f^1, f^2, ..., f^m],$$
$$x = c \ \star \ \{f^1, f^2, ..., f^m\}$$

where c is a generic chain specifier which can be existentially or universally quantified. In order to identify when a custom anomaly like "`Web traffic does not pass` through an IDS" is triggered in a case scenario like the network shown in Figure 1, we can use the following anomaly expressed in the two available representations:

$$usr\_src = Alice \land usr\_dst = Google \land trf\_type = \{http\}$$
$$\land c^k \not\succ [<*, IDS>], \forall c^k \in C \rightarrow NoWeb2IDS(r)$$

$$eth\_src = * \land eth\_dst = * \land eth\_type = 0\mathbf{x}0800 \land vlan\_id$$
$$= * \land ip\_src = 130.192.225.116 \land ip\_dst = 8.8.8.0/24 \land$$
$$ip\_proto = 0\mathbf{x}06 \land port\_src = * \land port\_dst = 80 \land$$
$$c^k \not\succ [<*, IDS>], \forall c^k \in C \rightarrow NoWeb2IDS(r)$$

6.1.4. Node Ordering anomalies

this class contains anomalies that are violations of ordering constraints on the functions traversed by a flow. Such constraints can be expressed in terms of the position of the w-th network function within a chain c (i.e., $\pi(f^w, c)$) and they may be required to hold for at least one or for all the chains of the forwarding rules that manage that flow. Of course, in order to express the flow for which the constraint is checked, network field comparisons can be used. Hence these anomalies are specified by formulas including the following comparisons:

$$x = n \ \star \ v, \qquad x = \pi(f^w, c) \ \star \ \pi(f^q, c) \quad \text{with } f^w, f^q \in c$$

An example is when we want to ensure that a NAT is always configured to process traffic before a firewall. This means that we have to detect the anomalous situation when a NAT is located after a firewall in the SFC topology, which can be done by the following anomaly definition based on the position operator:

$$usr\_src =^* \land usr\_dst =^* \land trf\_type =^* \land$$
$$\pi(<*, NAT>, c) > \pi(<*, FW>, c),$$
$$\exists c \in C \rightarrow NatAfterFW(r)$$

the equivalent formula in the header-based representation:

$$eth\_src =^* \land eth\_dst =^* \land eth\_type =^* \land vlan\_id =^* \land$$
$$ip\_src =^* \land ip\_dst =^* \land ip\_proto =^* \land port\_src =^* \land$$
$$port\_dst =^* \land \pi(<*, NAT>, c) > \pi(<*, FW>, c),$$
$$\exists c \in C \rightarrow NatAfterFW(r)$$
$$\tag{10}$$

Furthermore, another example is when we want to ensure that all traffic to web servers passes through web caches. The formula can be expressed as follows in the header-based representation:

$$\begin{aligned}&eth\_src =^{*} \wedge eth\_dst =^{*} \wedge eth\_type = 0\mathbf{x}0800\\ &\wedge vlan\_id =^{*} \wedge ip\_src =^{*} \wedge ip\_dst =^{*} \wedge\\ &ip\_proto = 0\mathbf{x}06 \wedge port\_src =^{*} \wedge port\_dst = 80 \wedge\\ &\pi(<*, SERVER>, c) < \pi(<*, CACHE>, c),\\ &\exists c \in C \rightarrow ServerBeforeCache(r)\end{aligned} \tag{11}$$

6.1.5. Chain Constraint anomalies

This category includes anomalies that can be detected by comparing the chains of a set with one another. (e.g., some chains in a forwarding rule are equal). Thus such anomalies contain comparisons between SFCs ($c^k \star c^w$) that belong to the same forwarding rule, and network fields comparisons to identify the flow ($n \star v$ or $n^o \star n^p$):

$$x = n \star v, \quad x = n^o \star n^p, \quad x = c^k \star c^l$$

As an example, let us consider a network graph where that a web traffic is balanced on two chains (Figure 1) and let us assume that we want to require that web traffic is processed either by the same (i.e., equivalent) chains or by a similar (i.e. correlated or dominated) chains. This means that the two chains must not be disjoint and we can detect this anomalous situation by means of the following header-based formula:

$$\begin{aligned}&eth\_src =^{*} \wedge eth\_dst =^{*} \wedge eth\_type = 0\mathbf{x}0800 \wedge\\ &ip\_src = 130.192.225.116 \wedge ip\_dst = 8.8.8.0/24 \wedge\\ &ip\_proto = 0\mathbf{x}06 \wedge port\_src =^{*} \wedge port\_dst = 80 \wedge\\ &vlan\_id =^{*} \wedge c^k \perp c^l, \forall c^k, c^l \in C \rightarrow DisjointChains(r)\end{aligned}$$

6.1.6. Sub-Optimization anomalies

Such anomalies aim at detecting under-optimizations of the policy specification and thus situations where more forwarding rules can be substituted by a single rule. In order to detect such anomalies, it is necessary to discover the forwarding rules that have the same sets of chains and properties. Hence the anomalies in this class include the following comparisons:

$$x = n_i^o \star n_j^p \quad x = c_i^k \in C_j, \forall c_i^k \in C_i \ \wedge c_j^l \in C_i, \forall c_j^l \in C_j$$

Under this class, we include the duplication anomaly defined in (4). The following formulas are another example, where we detect those forwarding rules that refer to completely disjoint traffic flows but that enforce the same set of SFCs:

$$\begin{aligned}&usr\_src_i \neq usr\_src_j \wedge usr\_dst_i \neq usr\_dst_j \wedge\\ &trf\_type_i \perp trf\_type_j \wedge c_i^k \in C_j, \forall c_i^k \in C_i \wedge\\ &c_j^l \in C_i, \forall c_j^l \in C_j \wedge ban_i = ban_j \wedge mem_i = mem_j \wedge\\ &ram_i = ram_j \wedge ncpu_i = ncpu_j \rightarrow SubOptimizedFlows(r_i, r_j)\end{aligned}$$

$$\begin{aligned}&eth\_src_i \neq eth\_src_j \wedge eth\_dst_i \neq eth\_src_j \wedge\\ &eth\_type_i \neq eth\_type_j \wedge vlan\_id_i \neq vlan\_id_j \wedge\\ &ip\_src_i \perp ip\_src_j \wedge ip\_dst_i \perp ip\_dst_j \wedge\\ &ip\_proto_i \neq ip\_proto_j \wedge port\_src_i \perp port\_src_j \wedge\\ &port\_dst_i \perp port\_dst_j \wedge c_i^k \in C_j, \forall c_i^k \in C_i \wedge\\ &c_j^l \in C_i, \forall c_j^l \in C_j \wedge ban_i = ban_j \wedge mem_i = mem_j \wedge\\ &ram_i = ram_j \wedge ncpu_i = ncpu_j \rightarrow SubOptimizedFlows(r_i, r_j)\end{aligned}$$

6.1.7. Conflicting anomalies

In our model, conflicts arise when two forwarding rules manage the same traffic flow but they do not specify the same sets of chains. If the two rules are installed into the network, inconsistencies in the traffic forwarding can be generated at run-time. Hence a formula for detecting this kind of anomaly includes the following comparisons:

$$\begin{aligned}&x = n_i^o \star n_j^p\\ &x = c_i^k \in C_j, \forall c_i^k \in C_i \quad x = c_i^k \notin C_j, \forall c_i^k \in C_i\\ &x = c_i^k \in C_j, \exists c_i^k \in C_i \quad x = c_i^k \notin C_j, \exists c_i^k \in C_i\end{aligned} \tag{12}$$

The comparisons x that compose a conflicting anomaly have been selected so as to enable the specification of different types of relationships between two sets of chains. An example of relationship is the case in which two sets C_i and C_j contain the same SFCs or also when C_i contains all the chains of C_j as subset. Another case is when C_i and C_j do not have any chain in common. This means that the policy contains two forwarding rules that forward the same traffic flow to different sets of SFCs. This kind of conflictual anomaly can be detected for example by the following formula:

$$\begin{aligned}
&eth\_src_i = eth\_src_j \wedge eth\_dst_i = eth\_dst_j \wedge \\
&eth\_type_i = eth\_type_j \wedge vlan\_id_i = vlan\_id_j \wedge \\
&ip\_src_i = ip\_src_j \wedge ip\_dst_i = ip\_dst_j \wedge \\
&ip\_proto_i = ip\_proto_j \wedge port\_src_i = port\_src_j \wedge \\
&port\_dst_i = port\_dst_j \wedge c_i^k \notin C_j, \forall c_i^k \in C_i \wedge \\
&c_j^l \notin C_i, \forall c_j^l \in C_j \rightarrow DisjointChains(r_i, r_j)
\end{aligned} \tag{13}$$

Note that some cases of “conflicting" forwarding rules according to the anomaly model (12) may not be considered by the network operators as conflicting anomalies. This is because this kind of anomaly depends on the network topology and on what the operator considers erroneous for her network. Let us consider the case of two forwarding rules r_i and r_j to forward the traffic between the end-host h_a and the web server ws_a:

$$\begin{aligned}
&r_i = ((eth\_src =^*, eth\_dst =^*, eth\_type =^*, vlan\_id =^*, \\
&ip\_src = 130.192.225.11, ip\_dst = 8.8.8.11, ip\_proto = 0x06, \\
&port\_src =^*, port\_dst =^*), \{[< h\_a, H >, < vpn\_a, VPN >, \\
&< fw, FW >, < ws\_a, WS >], [< h\_a, H >, < vpn\_b, VPN >, \\
&< dpi, DPI >, < fw, FW >, < ws\_a, WS >]\} \\
&ban = 100MB/s, mem = *, ram = 16GB, ncpu = 5) \\
&r_j = ((eth\_src =^*, eth\_dst =^*, eth\_type =^*, vlan\_id =^*, \\
&ip\_src = 130.192.225.11, ip\_dst = 8.8.8.11, ip\_proto = 0x06, \\
&port\_src =^*, port\_dst =^*), \{[< h\_a, H >, < vpn\_a, VPN >, \\
&< fw, FW >, < ws\_a, WS >], [< h\_a, H >, < vpn\_b, VPN >, \\
&< dpi, DPI >, < fw, FW >, < ws_a, WS >], [< h\_a, H >, \\
&< vpn\_a, VPN >, < mn, MN >, < ws\_a, WS >]\} \\
&ban = 100MB/s, mem = *, ram = 16GB, ncpu = 5)
\end{aligned}$$

In this example, r_j contains an additional SFC with respect to r_i, but this kind of policy (even if it is ambiguous and non-optimized) may not be a conflict because, for example, each of those chains contains a VPN functionality and the operator does not want to be advertised in such cases.

In this paper, we consider as pre-defined conflictual anomaly only the case when the two forwarding rules do not have any SFC in common, as defined in (13). All the other possible conflictual anomalies have to be specified by the network operators and are classified as custom anomalies (Table 4). An example of operator-defined conflicting anomaly could be the case of a traffic flow that is managed by two forwarding rules that enforce “correlated" sets of SFCs (i.e., the two sets share some SFCs but they are not the same):

$$\begin{aligned}
&eth\_src_i = eth\_src_j \wedge eth\_dst_i = eth\_dst_j \wedge \\
&eth\_type_i = eth\_type_j \wedge vlan\_id_i = vlan\_id_j \wedge \\
&ip\_src_i = ip\_src_j \wedge ip\_dst_i = ip\_dst_j \wedge \\
&ip\_proto_i = ip\_proto_j \wedge port\_src_i = port\_src_j \wedge \\
&port\_dst_i = port\_dst_j \wedge \exists c_i^k \in C_i, c_i^k \notin C_j \wedge \\
&\exists c_i^y \in C_i, c_i^y \in C_j \wedge \exists c_j^l \in C_j, c_j^l \notin C_i \\
&\exists c_j^p \in C_j, c_j^p \in C_i \rightarrow CorrelatedChains(r_i, r_j)
\end{aligned} \tag{14}$$

6.1.8. Global Properties anomalies

These anomalies arise when any flow violates some requirements specified by the network administrator about properties. Thus, a comparison x that is used to express this type of anomaly is takes the form: $x = p \star v_p$

An example of anomaly belonging to this class is when the network administrator wants that any flow consumes less than 1000 Mbit/s of bandwidth: in this case, any policy requiring more than 1000 Mbit/s for a flow is anomalous. Such anomalies are considered as custom anomalies.

6.1.9. *Specific Properties anomalies*

These anomalies are specific cases of the previous ones. They arise when one or more *specific* chain (i.e. one that contains a specific functions) traversed by $\mathcal{M}$, violates some requirements specified by the network administrator. The forms of comparisons in these anomalies are:

$$x = n \star v, \quad x = p \star v_p,$$
$$x = c \star [f^1, f^2, ..., f^m],$$
$$x = c \star \{f^1, f^2, ..., f^m\}$$

These anomalies belong to custom anomalies.

An example is when all chains traversed by a *http* flow, does not have more than 1TB of Memory allocated for that flow. The violation of this requirement can be formalized as the following anomaly:

$$trf\_type = \{http\} \land c^k \in [< *, CDN >], \forall c^k \in C \land$$
$$mem < 1TB \rightarrow CDNmemory(r)$$

A forwarding rule like the following one is affected by the above anomaly:

$$r = ((usr\_src = ^*, usr\_dst = ^*, trf\_type = http),$$
$$[< fw_a, FW >, < cdn_a, CDN >], (ban = 100MB/s,$$
$$mem = 2TB, ram = 16GB, ncpu = 3))$$

7. Implementation and performance

In order to evaluate our approach, we have implemented a Java-based prototype and tested its performance under different scenarios. We have run our tests on an Intel i7-4600U@2.10GHz workstation with 8 GB of RAM. The main purposes of this experimental evaluation were: *(i)* validate our model in a real-case scenario; *(ii)* identify the main factors that influence the verification performance; and finally *(iii)* evaluate verification time as a function of the most influential factors.

7.1. *Implementation*

Our Java-based prototype exploits Drools [26], as verification engine. Drools is a Rule-Based System that uses the ReteOO algorithm to perform the inferences [27]. A Rule-Based System is a Knowledge-Based System that encodes information in the form of rules. Listing 1 shows the structure of generic rules in the Drools language.

In our prototype, every Drools rule represents an anomaly, while forwarding rules are implemented as Java objects against which Drools rules are checked.

```
when
//conditions (query language)
then
//actions (java)
end
```

Listing 1: Structure of a rule statement in the Drools language.

As an example, Listing 2 contains a rule used to check if a forwarding rule presents a wrong source port definition (i.e., there is a source port anomaly).

```
rule "SRC_Port_anomaly"                                1
// Rule: wrong source port definition                  2
when                                                   3
  f : ForwardingRule(                                  4
      (getPortSrc().getStartValue()<0))                5
 as : AnomalySet()                                     6
 not(Anomaly(                                          7
       getType().compareTo("src_port")==0 &&           8
       getRuleId().contains(f.getId())                 9
             )from as.getAnomalies())                  10
then                                                   11
 Anomaly a=new Anomaly();                              12
 a.setType("src_port");                                13
 a.getRuleId().add(f.getId());                         14
 s.getAnomalies().add(a);                              15
end                                                    16
```

Listing 2: An example of Drools rule.

In other words, the meaning of Rule in Listing 2 is:
When:

- there is a forwarding rule f with source port less then zero (lines 4 to 5);

- in the anomaly set *as* (lines 6 and 10) there are no other anomalies, of type *wrong source port* (line 7), related to f (line 8).

Then:

- a new *wrong source port* anomaly related to f is created (lines 11 to 14).

7.2. Validation and Performance Evaluation

For validation, we used a sample network scenario obtained, by approximating part of our campus network. Such scenario (shown in Figure 2) has been manually setup with real data and policy rules. It contains about 35 clients, 15 servers, 10 network functions (i.e an IDS, a VPN Gateway, an Application Firewall, a Monitor, a Packet Filter, a Web Cache, an Anti Spam and two Load Balancers), and the policy contains 23 forwarding rules. In this scenario, we performed the validation in less than 5ms, by detecting 4 anomalies: 2 Single-Field, 1 Conflicting, 1 Node ordering.

In order to evaluate the performance of the proposed approach, some tests were run using a number of synthetically generated scenarios, so as to have a rough estimation of processing times in different cases.

Test scenarios were generated starting with a medium-size network, which is able to stress the main aspects of the solution. Specifically, this network includes about 300 hosts that generate several types of traffic flows (i.e., HTTP, POP3 and SMTP) towards the internal servers and the Internet, processed by about 10 network functions.

The forwarding rules used for the evaluation are expressed in the header-based representation, which allows a level of detail higher than the name-based representation and facilitates us in creating more complex anomalies in the testing scenario. A set of forwarding rules has been automatically generated to create each use-case. Thanks to the header-based representation, we have increased progressively the size of the forwarding rule set, by increasing the number of traffic flows and by considering wider and wider ranges of port numbers and traffic types and more subnets and hosts. In this way, we have been able to test the scalability of the verification process.

For what concerns the set of anomalies checked at each test-run, we have considered both those anomalies the model supports by default (i.e., the pre-defined set that includes 16 anomalies) and other new custom anomalies, specific for the tested network scenario. In detail, our forwarding rule set has been generated so as to trigger at least one anomaly for each class presented in Section 6.1. The whole set of anomalies that have been checked in our network scenarios is summarized in Tables 3 and 4, where, for each anomaly, we present a possible formula that detects that anomaly for a specific flow.

Moreover, in the automatic generation process, we have set a threshold on the percentage of forwarding rules (with respect to the total rule set) that trigger an anomaly. Figure 3a shows that, for each rule set-size, we have evaluated the elapsed time with the following percentages of anomalous rules: 10%, 20%, 50% and 80%.

The obtained results indicate that the elapsed time to complete verification grows linearly with the number of forwarding rules. This is highlighted in the four test scenarios. Each measured time has have been averaged on 100 test-runs. Verification time grows up to 340ms in the worst case (the solid line in Figure 3a).

In order to also evaluate the dependency of verification time on the percentage of forwarding rules that trigger an anomaly, we report another plot (Figure 3b). Keeping constant the number of forwarding rules, we plot verification time for percentages of anomalies growing from 0% to 100%. The plot in Figure 3b, shows the behavior for a number of forwarding rules set to 100, 300, 500, 700 and 1000 rules. Once again the dependency is linear.

As we can note from the achieved results (Figure 3b), the percentage of forwarding rules that bring to an anomaly has a greater influence on verification time when the rule

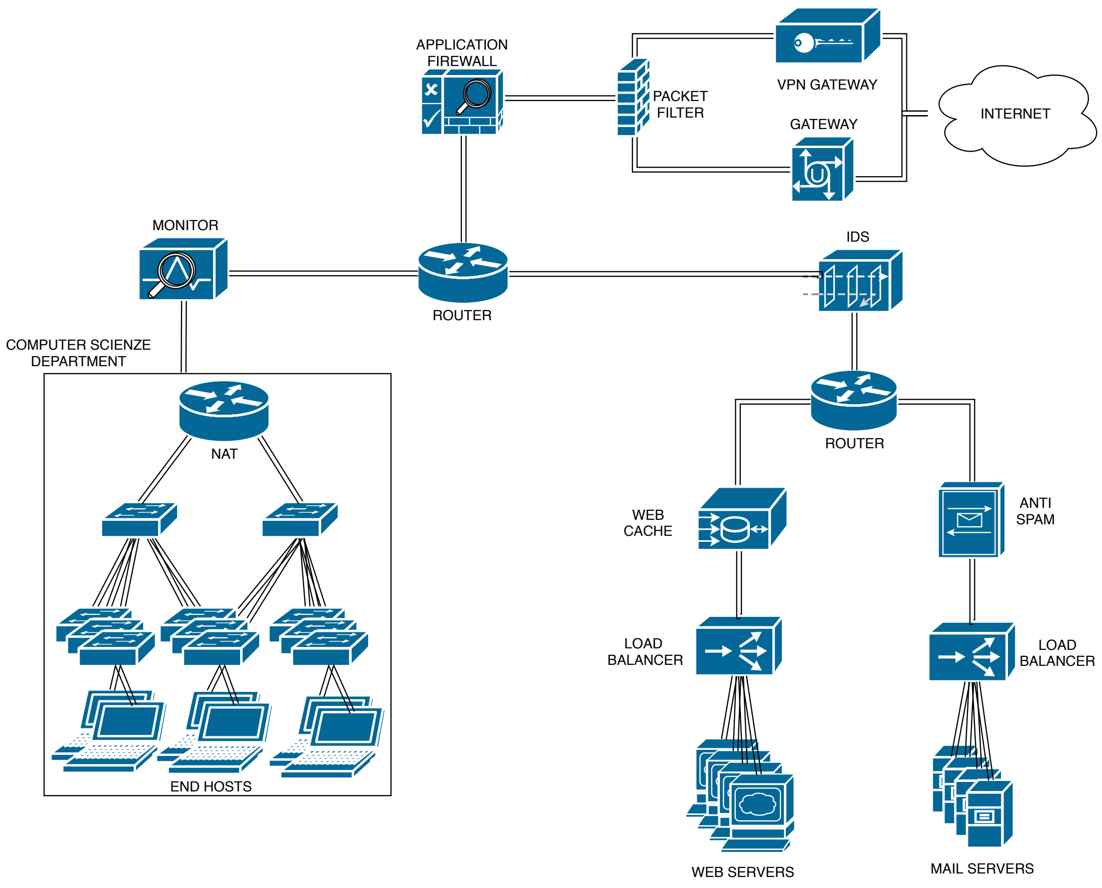


Figure 2: Campus network topology.

Table 3: Pre-defined set of anomalies.

Class	Anomaly	Formula
Single-Field	bad *usr_src* specification	see (5)
	bad *port_src* specification	see (6)
	bad *port_dst* specification	see (7)
	bad *vlan_id* specification	$vlan\_id < 1 \wedge vland\_id > 4094 \rightarrow BadVlanId(r)$
	bad *eth_type* specification	$eth\_type \neq \{0x0800, 0x0806, 0x8100\} \rightarrow BadEthType(r)$
	bad *ip_proto* specification	$ip\_proto \neq \{0x01, 0x06, 0x11\} \rightarrow BadIpProto(r)$
Pair-Field	equal source and destination names	see (8)
	equal source and destination IP addresses	see (9)
	equal source and destination Ethernet addresses	$eth\_src = eth\_dst \rightarrow BadEthernetAddress(r)$
Sub-Optimization	forwarding rule duplication	see (4)
Conflicting	a single flow is forwarded to different chains	see (13)

set size grows. This can be confirmed by comparing the trend in the case of 100 forwarding rules, where the elapsed time is almost constant, and in the case of 1000 rules, where the elapsed time grows more rapidly with the increment of the anomaly percentage.

The achieved results are also confirmed in an additional

Table 4: Others custom set of anomalies.

Class	Anomaly	Formula
Node Traversal	web traffic does not traverse a web-cache	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.116 \wedge ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 80 \wedge c^k \nsucc [< *, CACHE >], \forall c^k \in C \rightarrow WebNot2Cache(r)$
	mail traffic does not traverse an anti-spam	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.244 \wedge ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 25 \wedge c^k \nsucc [< *, SPAM >], \forall c^k \in C \rightarrow MailNot2Spam(r)$
	web traffic traverses an anti-spam	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.116 \wedge ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 80 \wedge c^k \succ [< *, SPAM >], \forall c^k \in C \rightarrow Web2Spam(r)$
	mail traffic traverses a web-cache	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.244 \wedge ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 25 \wedge c^k \nsucc [< *, CACHE >], \forall c^k \in C \rightarrow Mail2Cache(r)$
	internet traffic does not pass through a L7 firewall	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 80 \wedge c^k \nsucc [< *, L7\_FW >], \forall c^k \in C \rightarrow InternetNot2Firewall(r)$
Node Ordering	firewall non located after a NAT function but before it	see (11)
Chain Constraint	Internet traffic is not forwarded to correlated chains	$eth\_type = 0\mathbf{x}0800 \wedge ip\_src = 130.192.225.116 \wedge ip\_dst = 8.8.8.0/24 \wedge ip\_proto = 0\mathbf{x}06 \wedge port\_dst = 80 \wedge c^k \nsim c^l, \forall c^k, c^l \in C \rightarrow InternetNoCorrelatedChains(r)$
Conflicting	a single flow is forwarded to different chains	see (14)

test case (Figure 3c), where we have evaluated verification time with a growing number of "anomalous" rules in different sized rule-sets (i.e., 100, 300, 500, 700 and 1000 forwarding rules). Also in this test-scenario, it is evident that the performance of our verification approach is influenced by both the number of forwarding rules and the percentage of these that trigger an anomaly.

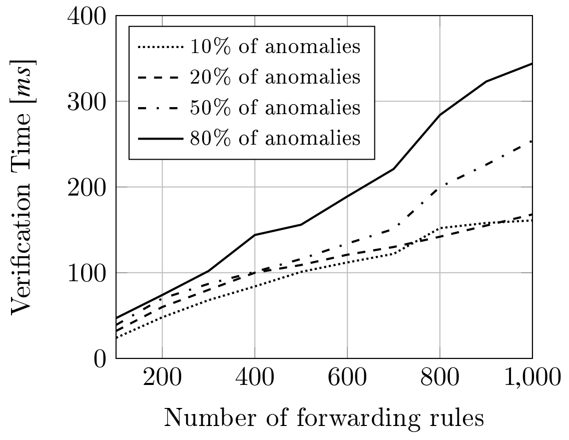
Moreover, we can also note that the verification time is in the range between 340ms, in the worst case with 1000 forwarding rules and 80% of "anomalous" rules (Figure 3a), and 400ms, when each one of the 1000 forwarding rules triggers an anomaly (i.e., the solid lines in Figure 3b and Figure 3c).

The achieved results show that our verification approach takes a time in the order of hundreds of milliseconds in the case of a real-sized network with a growing number of traffic flows and time increases linearly with the complexity. Moreover, we measured the memory required by our tool during the execution of synthetically generated scenarios. We noted that the memory consumption was approximately the same in all scenario and in the worst case (i.e., 1000 forwarding rules with 80% of anomalies) it was approximately 1GB.

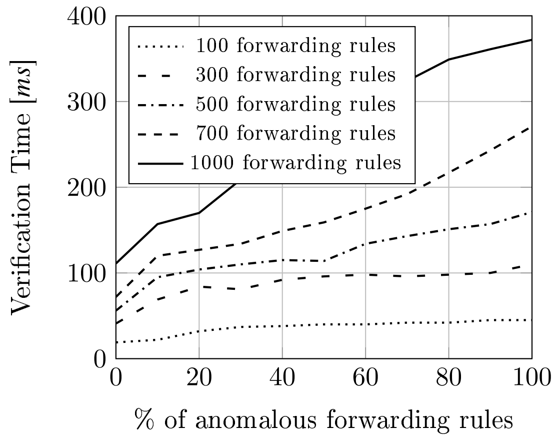
Considering the performance in terms of time and memory, it is reasonable to use our approach in a real network scenario.

8. Conclusion

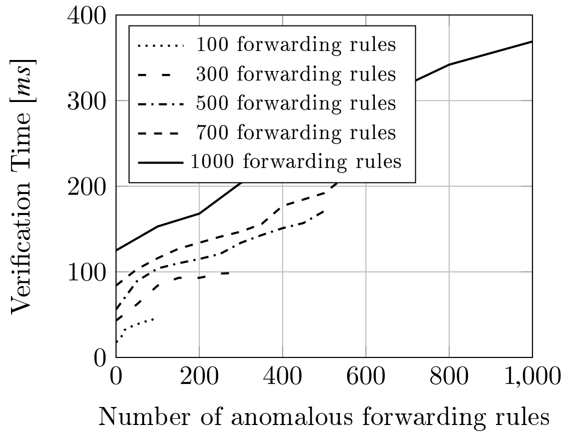
In this paper we have proposed a formal approach to specify and verify SFC policies. According to the proposed approach, the presence of anomalies in a forwarding policy can be detected before deployment, i.e. before the policy rules are enforced by the SDN Controller and installed into the network switches. In order to achieve this goal, we have designed a two-fold formal representation of the forwarding policy that characterises packet forwarding in the network (i.e., in terms of traffic flows and service chains) and of the set of anomalies that have to be detected against the



(a) Verification time evaluated with a growing number of forwarding rules.



(b) Verification time evaluated with a growing percentage of forwarding rules that satisfy an anomaly.



(c) Verification time evaluated with a growing number of anomalies.

Figure 3: Evaluation times of forwarding policy verification.

policy rules. This has been done using standard notations such as First Order logic and Horn clauses.

This formal approach enables precise and unambiguous specifications of policy rules and of related anomalies, and, through the application of already existing verification engines, it allows rigorous verification of the absence of anomalies and the consequent guarantee that a verified policy is anomaly-free, under the assumption that the verification engine is correct.

Moreover, the proposed model is highly flexible and extendible, because it allows network operators to define their own sets of anomalies. A minimum level of correctness in the network can be always guaranteed, by having a core set of pre-defined anomalies (e.g., capturing bad policy rule specifications or forwarding loops), which can then be extended by the operators.

In order to prove the usefulness of this approach in a real network scenario, we have implemented and tested a Java-based prototype of our verification model, exploiting Drools as inference engine. We have achieved verification times in the magnitude of milliseconds for networks of reasonable size. This evaluation has been performed under different conditions created by increasing the number of forwarding rules configured in the considered network use-case and the percentage of such rules that trigger an anomaly.

For the future, we plan to extend the expressiveness of the model by considering also the configurations installed into the network functions that make up the service chains. In some cases, flow forwarding may depend also on the packet processing performed by the network functions, which, in turn, depends on the function configurations.

Moreover, in order to further improve the usability of the approach, in the future we aim to extend the proposed approach by exploiting techniques for the automatic refinement of anomaly rules, starting from high-level requirements, and automatic strategies for the update of policy rules when anomalies are detected.

Finally, the proposed model could become a wider, and more ambitious contribution. Since policy-based systems are largely widespread in data protection, filtering, access control, and many other policy domains, a useful contribution can be to extend this verification model in order to encompass different policy domains. An extended verification model could verify that a domain-specific policy is consistent also in the presence of policies belonging to other domains.

ACKNOWLEDGMENT

This work was supported in part by the European Commission, under Grant Agreement no. 786922.

[1] J. Halpern and C. Pignataro, "Service function chaining (sfc) architecture," Internet Requests for Comments, RFC Editor, RFC 7665, 2015.

[2] R. Sahay, W. Meng, and C. D. Jensen, "The application of software defined networking on securing computer networks: A survey," *Journal of Network and Computer Applications*, vol. 131, pp. 89 – 108, 2019.

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

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

[5] S. K. Dey, M. M. Rahman, and M. R. Uddin, "Detection of flow based anomaly in openflow controller: Machine learning approach in software defined networking," in *2018 4th International Conference on Electrical Engineering and Information Communication Technology (iCEEiCT)*, Sep. 2018, pp. 416–421.

[6] E. Tantar, A.-A. Tantar, M. Kantor, and T. Engel, "On using cognition for anomaly detection in sdn," in *EVOLVE - A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation VI*, A.-A. Tantar, E. Tantar, M. Emmerich, P. Legrand, L. Alboaie, and H. Luchian, Eds. Cham: Springer International Publishing, 2018, pp. 67–81.

[7] M. Cheminod, L. Durante, L. Seno, F. Valenza, A. Valenzano, and C. Zunino, "Leveraging sdn to improve security in industrial networks," in *2017 IEEE 13th International Workshop on Factory Communication Systems (WFCS)*, May 2017, pp. 1–7.

[8] 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.

[9] C. Basile, D. Canavese, C. Pitscheider, A. Lioy, and F. Valenza, "Assessing network authorization policies via reachability analysis," *Comput. Electr. Eng.*, vol. 64, no. C, pp. 110–131, 2017.

[10] 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. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934881

[11] R. Bifulco and F. Schneider, "Openflow rules interactions: definition and detection," in *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013, pp. 1–6.

[12] 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 (NSDI'12)*. USENIX Association, 2012, pp. 10–10.

[13] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 290–301, oct 2011.

[14] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *ACM SIGPLAN Notices*, vol. 46, no. 9. ACM, 2011, pp. 279–291.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, no. 1, ACM. ACM Press, 2012, pp. 217–230.

[16] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," *Technical Report of USENIX*, 2013.

[17] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster, "Managing the network with merlin," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 24.

[18] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 10, pp. 2069–2084, 2005.

[19] F. Valenza, C. Basile, D. Canavese, and A. Lioy, "Classification and analysis of communication protection policy anomalies,"

IEEE/ACM Transactions on Networking, vol. 25, no. 5, pp. 2601–2614, Oct 2017.

[20] J. G. Alfaro, N. Boulahia-Cuppens, and F. Cuppens, "Complete analysis of configuration rules to guarantee reliable network security policies," *International Journal of Information Security*, vol. 7, no. 2, pp. 103–122, 2007.

[21] M. G. Gouda and A. X. Liu, "Structured firewall design," *Computer Networks*, vol. 51, no. 4, pp. 1106–1120, mar 2007.

[22] B. Lopes Alcantara Batista, G. A. Lima de Campos, and M. P. Fernandez, "Flow-based conflict detection in openflow networks using first-order logic," in *Proceedings of the IEEE Symposium on Computers and Communication (ISCC 2014)*. IEEE, mar 2014, pp. 1–6.

[23] C. Prakash, J. Lee, Y. Turner, J. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "PGA: using graphs to express and automatically reconcile network policies," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, 2015, pp. 29–42.

[24] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, Sep. 2016.

[25] A. Fischer, J. F. Botero, M. Duelli, D. Schlosser, X. Hesselbach, and H. de Meer, "ALEVIN - A framework to develop, compare, and analyze virtual network embedding algorithms," *ECEASST*, vol. 37, 2011.

[26] M. Bali, *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing Ltd, 2009.

[27] D. Sottara, P. Mello, and M. Proctor, "A Configurable Rete-OO Engine for Reasoning with Different Types of Imperfect Information," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 11, pp. 1535–1548, 2010. [Online]. Available: https://doi.org/10.1109/TKDE.2010.125

Fulvio Valenza (fulvio.valenza@polito.it) received his M.Sc. (summa cum laude) in 2013 and his Ph.D. (summa cum laude) in Computer Engineering in 2017 from the Politecnico di Torino, Torino, Italy. Currently, he is a Researcher at the Politecnico Torino, Italy. His research activity focuses on network security policies, orchestration and management of network security functions in SDN/NFV-based networks, threat modelling.

Serena Spinoso (serena.spinoso@polito.it) received her Ph.D in Computer and Control Engineering from Politecnico di Torino, Turin, Italy in 2017 and her M.Sc.Degree (summa cum laude) in Computer Engineering in 2013. 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.

Riccardo Sisto (riccardo.sisto@polito.it), MS in Electronic Engineering in 1987, Ph.D in Computer Engineering in 1992, has been with Politecnico di Torino as researcher, associate professor, and, since 2004, full professor of Computer Engineering. His current research activity is about formal methods applied to computer networks. On this and related topics he has authored and co-authored more than 100 scientific papers. Riccardo Sisto is senior member of the ACM.