A Two-Fold Traffic Flow Model for Network Security Management

A two-fold traffic flow model for network security management

Daniele Bringhenti, Simone Bussa, Riccardo Sisto, Fulvio Valenza

***Abstract*—Introducing formal methods in the automatic resolution of network security management problems can guarantee solution correctness, so also boosting human confidence in using automatic techniques. A necessary step to achieve this feature is the definition of formal network models, representing network topology, traffic flows, etc. Each state-of-the-art formal network modeling approach has been proposed and validated only for a specific management problem (e.g., verification of configurations or refinement of policies into configurations). This paper analyzes a possible combination of the most promising state-of-the-art modeling approaches into a unified formal model that can be used by existing automatic resolution algorithms to solve both the verification and the refinement problems, without the need of major changes. The model is flexible enough to allow different aggregation levels of traffic into flows. The paper analyzes two opposite flow aggregation strategies, named Atomic Flows and Maximal Flows, and compares their performance when applied to the two identified security problems.**

***Index Terms*—network security management, policy-based management, network formal models**

I. Introduction

In the last decade, computer networks have been significantly reshaped by the advent of softwarization paradigms, such as *Software-Defined Networking* (SDN) and *Network Functions Virtualization* (NFV). Nowadays, network security management must comply with the agility and dynamism characterizing virtual environments [1]. In order to meet this requirement, the traditional security management approaches, based on manual trial-and-error techniques, are being progressively abandoned, because they are cumbersome, time-consuming, and error-prone. In particular, they are being replaced by automatic processes and tools, which can exploit a comprehensive view of the network to simplify and speed up security management operations, while reducing the number of fallible human interventions.

A possible paradigm through which automation can be introduced into network security is policy-based management [2], which lately in literature is also tending to intent-based networking [3]. The core idea is that human administrators should just specify the security requirements or intents to be managed in a network by means of sentences expressed in a user-friendly language, and then automatic resolution algorithms are employed to solve the related network security management problems [4] (e.g., to verify if an existing security configuration fully satisfies the required policies, or to automatically derive the configuration from them). A key feature of employing policy-based management for network security automation is that it is suitable to be paired with formal methods. This feature can assist in guaranteeing that the automatically computed results of the policy-based management operations are actually correct and compliant with the security requirements to be enforced in the network [5]. Therefore, it increases the human confidence in employing automated tools and it contributes to the success of network security automation.

In this context, a central research activity is the investigation of adequate network models, representing the key elements for the application of policy-based security management, as well as for other automatic techniques (e.g., approaches based on artificial intelligence). A concept that plays a central role in such models is the representation of network traffic flows. Formally modeling how the packets originating from the source of a communication can be forwarded and modified in a network until they reach their destination is a complex task, which involves modeling multiple elements: (i) the packets that can cross a network; (ii) the paths that each packet can follow in the network; and (iii) the transformations that each network function can produce in the packets that traverse it.

Research in formal network models has made progress over the last years. The first models proposed in literature for verification purposes [6]–[8] could describe only simple networks, without functions that can modify packets, and were characterized by limited scalability. These limitations have been overcome by more recent approaches such as AP Verifier [9], its extension APT [10] and Verigraph2.0 [11] for verification, while ConfigSynth [12] and Verefoo [13] for refinement. However, most of these models have been designed to assist a single security management operation (verification or refinement), and they are usually not flexible enough to allow the application of the other operation. The only exception is the pair Verigraph2.0 - Verefoo, because these approaches share a similar network and flow modeling approach, which is based on modeling traffic flows rather than packet classes. In particular, those approaches aim to aggregate flows as much as possible, creating entities named Maximal Flows. Another interesting concept, which was introduced in AP Verifier and extended in APT to networks with transformers, is Atomic Predicates, representing the coarsest equivalence classes of packets in a particular network (i.e., two packets belong to the same class if they are not distinguished by any middlebox in the network). Even if this concept has been formulated and

D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza are with the Politecnico di Torino, Dip. Automatica e Informatica; e-mail: {first.last}@polito.it.

used only for the verification problem, it features very good performance and scalability.

In this paper, we aim to improve this state of the art about research in formal network models with multiple contributions. First, we present a new flow grouping strategy, aiming to create flow entities named Atomic Flows. This strategy leverages the concept of Atomic Predicates so as to have packet classes considered for traffic flow computation that are minimal and disjoint from each other. This feature allows a finer flow granularity than what Verigraph2.0 [11] and Verefoo [13] can achieve with Maximal Flows, despite creating a larger number of flows. In order to clarify all their differences, in this paper we fully detail not only the novel Atomic Flow strategy, but also the Maximal Flow strategy, as it was not completely described in the Verigraph2.0 [11] and Verefoo papers [13], whose scope was different. Then, we developed Verigraph2.0 and Verefoo variants based on Atomic Flows instead of Maximal Flows. These new variants of the two original frameworks allowed us to experimentally compare the efficacy and efficiency of the two algorithms for flow computation in the context of the two main operations related to policy-based management for network security, i.e., verification of connectivity properties (i.e., reachability and isolation properties) [14], and refinement (i.e., deriving the security configuration from the user-specified policies [15]). A main objective of this comparative analysis was to identify the flow aggregation strategy that is characterized by lower computation time and memory usage to solve each one of the analyzed security management problems.

The idea of this two-fold traffic flow model was presented preliminarily in [16]. This paper improves and completes that preliminary idea in the following ways: (i) the formalization of the traffic flow model, and the presentation of the two algorithms for computing Atomic Flows and Maximal Flows, so as to show the generality and flexibility of this model; (ii) a complete use case to which both algorithms are applied so that each step of them can be more easily understood; (iii) an extended discussion of related work to emphasize the novelty of this proposal; (iv) an improved experimental validation of the approach with application to realistic computer network topologies and the addition of new metrics in the scalability tests; (v) an extended comparison of the two algorithms with the identification of their respective advantages and disadvantages.

The remainder of this paper is structured as follows. Section II discusses related work. Section III describes the approach that is pursued for modeling and computing traffic flows. Section IV introduces the models of basic networking concepts (network topology, network functions, packets classes, security policies). Section V and Section VI illustrate the algorithms for the computation of Atomic Flows and Maximal Flows respectively, built on the previously presented models. Section VII describes how the modeling approach has been validated and the two algorithms have been compared, highlighting their advantages and drawbacks. Finally, Section VIII briefly concludes the paper and outlines future work.

II. Related Work

Several state-of-the-art approaches pairing security policy-based management with formal methods (i.e., [6]–[13], [17]–[25]) showed the need to model network packets or traffic flows, as they represent the key components for modeling more complex elements (e.g., the behavior of network functions, or the network security policies specified by the administrator). Here, our focus is on the formal models proposed to address the problems of verifying network security properties or of automating security configuration from user-specified policies. These two problems also represent the scenarios our model has been validated on. To this regard, Subsection II-A discusses the main characteristics and limitations of formal models proposed in literature to verify network security properties, whereas Subsection II-B focuses on the models defined in the context of automated approaches for network security configuration.

A. Flow models for verifying network security properties

Initially, the problem of defining traffic flow models for the verification of network security properties (e.g., network connectivity) was addressed without focusing on possible optimizations. In this respect, the milestone in this research area, i.e., [17], proposed an approach that computes and verifies the reachability a network can provide from a static snapshot of the configuration state from each of the routers composing the network topology. In doing so, it models all the possible packets whose reachability should be checked.

Later, some optimizations were introduced. On the one hand, [18] proposed a formal approach, named *Header Space Analysis* (HSA), based on a geometric model of the packet header space. Packets are modeled as points in the geometric space $\{0,1\}^L$, where L is the header length, and network functions as transfer functions on that space itself. Consequently, the operations performed on a packet along a path are obtained by composing the operations of the network functions it crosses. [19] extended this approach by adapting it to real-time property checking. On the other hand, Veriflow, the methodology illustrated in [6], combines packets into disjoint *Equivalent Classes* (ECs), i.e., sets of packets experiencing the same forwarding actions throughout the network. For each EC, Veriflow builds a forwarding graph representing the paths this EC can take according to the forwarding behavior of the network. However, both the approaches discussed in [18] and [6] initially model all the packets, before selecting or combining them into the classes that are really needed to check the requested security properties.

Other solutions are based on symbolic modeling, which avoids the explicit modeling of each packet. [20] proposes *Network Optimized Datalog* (NoD), a tool that models each packet header field as a separate symbolic variable, and a network function as a predicate that takes as parameters all the header fields in the packet. A symbolic execution explores all possible paths through the program, computing the possible values for each symbolic variable at every point. As not all such values are usually necessary, but only some of them, there is a redundancy of the results that need to be computed.

Another tool, named Symnet [21], tried to overcome this limitation by proposing the *Symbolic Execution Friendly Language* (SEFL). This programming language is used to express the data plane of the network so as to enable fast symbolic execution, as models are by construction memory safe, have bounded memory usage and are guaranteed to terminate. One of the limitations of this approach, as the authors state in [21], is that accurately modeling network functionalities that work with their proposed traffic model requires expertise from human users. However, they provide parsers only for a limited number of functions (i.e., switches, routers, and firewalls). This makes their models difficult to be extended to solve other security problems and for heterogeneous networks.

Among the most recent studies proposing traffic models for verifying network properties, two techniques are particularly relevant: Verigraph2.0 [11] and APT [10], which is an extended version of AP Verifier [9]. The former [11] models each packet class as a conjunction of predicates set over the packet fields, so that these predicates can be used for the formulation of a constraint programming problem to represent the verification problem, and it introduces the model of a traffic flow, which includes a collection of packets forwarded along the same path. The latter [10] introduces the concept of Atomic Predicate, used to represent a class of packets that are treated identically by every filter and transformer in the network. In their approach, the packet class represented by each Atomic Predicate can be identified by an integer, so that all the required operations of the verification process only deal with the integers that represent such classes. Nevertheless, each packet class is modeled as a bit array, and each predicate is a Boolean formula where each variable represents one bit in the packet header. In our view, this formalization may make it difficult to use this strategy for other security management operations, such as configuration, as the number of variables employed is high (i.e., a Boolean variable for each packet header bit).

Anyhow, we were inspired by the proposed concept of Atomic Predicate for the formulation of the algorithm to compute Atomic Flows (analyzed in Section V), casting that concept into a different traffic model based on less variables, as predicates are set over the packet fields instead that on their single bits.

B. Flow models for automatic network security configuration

Among the studies that apply formal methods to the automatic configuration of network security functions, the first ones, such as [7], [8] and [22], have limited models for network traffic. In particular, when mapping user-specified policies onto the configuration of security functions (mostly firewalls), these approaches do not take into account the possible transformations that may be applied to the traffic by functions such as NATs or load balancers.

Then, in more recent papers, such as [12], [13], [23]–[25], modeling flows has gained more interest, mainly due to the need to make automated methodologies applicable to distributed security functions, e.g. distributed firewalls, where it is necessary to have a global view of all the traffic crossing the network rather than of the packets reaching a single function instance. In greater detail, [12] simply identifies traffic flows by the identifiers of the end points of the communication and it uses them to decide how to secure a user-provided network graph by allocating different types of network security functions. Their flow model is rather simplistic, as it does not take into account the possible values that packet header fields such as IP addresses or ports may have. [23] also models the protocol that is used for the communication (e.g., HTTP, SSH), while [24] considers a large number of packet attributes (e.g., the IP 5-tuple fields or the timestamp) to represent each traffic flow, so that the methodology proposed in those papers takes decisions on the values of those attributes to automate the composition and configuration of chains composed of network security functions.

Finally, [13], [25] introduce a model which represents packets including all the fields of the IP 5-tuple, but open to include more fields, and traffic flows, modeled in a way similar to the one introduced in Verigraph2.0. Even if their traffic model is tailored to packet filtering and its validity for other security management problems is not proved, a relevant concept is that the approach proposed there tries to group multiple flows that behave in the same way (i.e., that cross the same node sequence and are subject to the same changes) into a single one.

C. Our contributions

All the studies mentioned above define network and traffic models to solve specific verification or configuration tasks, but they did not prove that the same models can be used efficiently to solve other security-related problems. Moreover, each method is affected by some limitations, as illustrated above.

This paper starts from the most promising modeling approaches proposed so far, and studies a possible way of combining them into a general formal model that can be used both for verification and refinement purposes, and that overcomes the most important limitations of the original techniques. More specifically, our contributions are the following ones:

- Differently from the modeling approaches used by Verigraph2.0 [11] and by Verefoo [13], where flows are joined together in order to reduce their number, here we also study a different grouping strategy, which we call Atomic Flows. This novel strategy is inspired by the APT approach of coarsest equivalence classes [10], but cast to the traffic flow model, using efficient field models rather than header bits.
- We formalize and validate the algorithm to compute Atomic Flows, according the Atomic Predicates strategy. Moreover, we provide a complete description of the Maximal Flows computation algorithm, which was not fully investigated in the Verigraph2.0 [11] and Verefoo [13] papers.
- We compare the two strategies with an extensive series of experimental tests to assess their performance, their

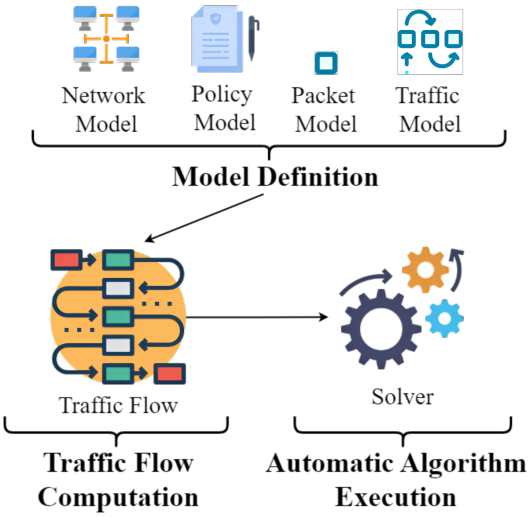


Fig. 1: The flow modeling approach

applicability to security management operations and to provide their trade-offs.

III. The Flow Modeling Approach

This section describes the workflow where the proposed flow modeling approach can be used to pair automation and formal methods for network security management. This workflow is composed of three main steps, as illustrated in Fig. 1:

1) the definition of general models for the main components of the network security management problem;
2) the computation of the traffic flows that cross the network under consideration;
3) the execution of an automatic resolution algorithm tailored to solve the security management problem.

A. Model definition

The first step of the approach consists in the definition of formal models for the main components of the network security problem. In particular, four main model classes are needed:

- The packet class model is needed to identify and group packets which belong to the same traffic flows and which are subject to the same networking operations.
- The network model consists in the representation of both the network topology structure and the forwarding and transformation behaviors of each function composing it. This model is needed to understand how packet classes cross the network and how they are managed by intermediate middleboxes.
- The traffic flow model is needed to provide a general representation for the flows of packets that cross the network, so as to apply the traffic flow algorithms to compute the flows that are interesting for the security policies.
- The network security policy model is needed to represent the requirements that must be managed in the network (e.g., that must be verified or enforced), so as to identify the network areas and the traffic flows that concern them.

In the definition of these models, a main aim has been to achieve a trade-off between adherence to reality and efficiency.

On the one hand, the models must capture all the information that may influence the correctness of the solution. If a critical piece of information is missing, the output of the automatic verification or resolution algorithm may not be correct, even if there is no issue in the algorithm design. For example, the configuration of a packet filtering firewall must be modeled so as to consider the decision that the function may take for any kind of possible input traffic (e.g., by modeling a default action that is applied whenever there is no match of the packet fields with the conditions of another filtering rule).

On the other hand, such models must be lightweight enough so as not to impact too much on the performance of the algorithms. If an excessive amount of redundant information is included in the models, their processing will require additional execution time, without altering the final result. For example, the behavior of a firewall should be modeled so as to capture only the features related to the security properties that must be enforced in a network (e.g., only the features related to forwarding operations if the security requirements concern reachability or isolation properties). Defining models that are lightweight enough is particularly important when the algorithm is based on a constraint-based programming problem, such as an integer linear programming problem or a *Satisfiability Modulo Theories* (SMT) problem. Less complex models allow reducing the number of variables employed for the problem formulation, and consequently the size of the solution space that the automated solver of such kind of problems must exhaustively explore.

The models that are defined in this paper to achieve such trade-offs are detailed in Section IV.

B. Traffic flow computation

The second step of the approach consists in computing all the traffic flows that are relevant for solving the network security management problems. For instance, if a human administrator wants to verify a reachability property, the flows that pertain that requirement must be identified and computed before the execution of the verification algorithm.

In our vision, a traffic flow represents how a packet class, generated by a network end point, crosses an ordered list of intermediate network functions before reaching the destination, and how these functions may transform the packets belonging to this class before possibly forwarding them to the next hop. According to this definition, the traffic flow computation may produce different results depending on how single network packets are grouped into classes. In particular, the choice of a grouping strategy has impact on the following main features: 1) the number of traffic flows that are computed; 2) the granularity level of each traffic flow, expressing how small the packet class associated with the flows is.

As different trade-offs can be reached for these characteristics, we have analyzed two different algorithms for traffic flow computation. Both of them work on the same general formal models proposed for packet classes and traffic flows, but they balance the number of computed flows and their granularity level in different ways. The flows that are computed with these two algorithms are respectively named Atomic Flows and Maximal Flows.

The computation of Atomic Flows is inspired by the idea of Atomic Predicates originally proposed in [9], as previously stated in Section II. According to that study, given a set of predicates on packet fields, it is possible to compute the set of totally disjoint and minimal predicates, named Atomic Predicates, such that each predicate can be expressed as a disjunction of a subset of atomic ones. Applying this concept to a computer network, it is possible to split each complex predicate (e.g., a predicate representing a firewall rule field, a NAT input packet class, or the condition of a security policy) into a disjunction of simpler and minimal Atomic Predicates. After computing the set of Atomic Predicates for all the predicates used in the network, the Atomic Flows can be computed so that each packet class that is associated with one of them is represented by an Atomic Predicate. This guarantees a fine granularity level, as each packet class considered for traffic flow computation is the minimal one, and is disjoint from the others. Another main advantage of this approach is that, as all Atomic Predicates are disjoint for definition, each packet class related to an Atomic Flow can be associated with an integer number that uniquely identifies it. This feature allows the execution of easier operations, both for traffic flow computation and for the resolution of the security management problem, because they can work on integers instead of complex predicates. Nevertheless, a drawback is that a larger number of traffic flows are produced with this algorithm. The algorithm for the computation of Atomic Flows is detailed in Section V.

Instead, the computation of Maximal Flows is based on the opposite criterion, which is to reduce the number of generated flows, while maximizing their aggregation. In this second case, all traffic flows that behave in the same way when crossing the network are grouped into a single Maximal Flow, so that it is sufficient to consider one Maximal Flow in the resolution of the security management problem instead of considering all the flows that make it up. The execution time for computing the Maximal Flows is expected to take less time, because the number of actual flows that must be computed is minimized. The counterbalance of such approach is that the packet classes whose transformation through the network is represented by a Maximal Flow have a coarser granularity, as they may represent a larger set of packets with respect to the approach based on the computation of Atomic Flows. Besides, these packet classes may not be disjoint, which complicates their use and requires that the packet classes they represent be preserved, instead of being neglected as it is possible with atomic flows. Indeed, differently from the approach based on Atomic Flows, the predicates representing packet classes cannot be uniquely identified by integer numbers. The algorithm for the computation of Maximal Flows is detailed in Section VI.

C. Automatic algorithm execution for security management

The third step of the approach consists in executing a specific automatic resolution algorithm for the network security management problem that needs to be addressed. State-of-the-art algorithms, already available in literature, may be employed for this step with minor changes. On the one hand, some changes are required to make those algorithm compliant with the formal models defined for the main components of the network security problem. However, as these models are designed to be as general as possible, most algorithms, already defined to work with custom network models, can be easily adapted to work with them. On the other hand, some modifications are due to the choice of which strategy is used for flow computation, as Atomic and Maximal Flows rely on partially different models for packet classes and traffic flows. Nevertheless, these changes are not related to the core approach of the selected resolution algorithm. Instead, they are mostly related to the actual low-level operations that must be executed (e.g., operations related to Atomic Flows work with integer numbers, while those related to Maximal Flows work with more complex predicates).

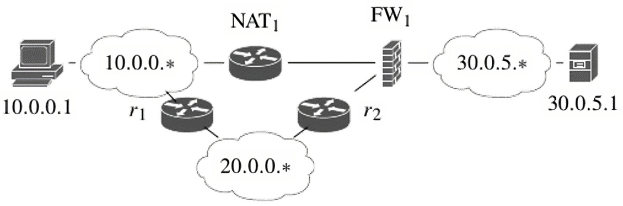
In order to show that this approach can be pursued by slightly modifying and extending existing resolution algorithms, we have applied it to two state-of-the-art algorithms for network security automation, available as open-source code: Verigraph2.0[1] [11] and Verefoo[2] [13]. The former automatically verifies connectivity properties (i.e., reachability and isolation properties) by providing formal assurance of the result correctness. Instead, the latter automatically refines user-specified security policies into the allocation scheme and configuration of distributed packet filtering firewalls, by formulating the security automation problem as a *Maximum Satisfiability Modulo Theorie*s (MaxSMT) problem so as to ensure correctness by construction for its output. Section VII shows how both these resolution strategies could be extended to support both Atomic and Maximal Flows, and how the two flow computation algorithms behave in terms of performance when applied to solve different network security management problems.

D. Running example

In order to aid the understanding of the formal models and of the traffic flow computation algorithms, Fig. 2 shows a possible scenario that will be used as a running example to describe the proposed approach in the next sections of the paper. The figure represents a network composed of two client sub-networks (10.0.0.0/24 and 20.0.0.0/24) and a server (30.0.5.1 in sub-network 30.0.5.0/24). The network includes also a NAT (30.0.0.1) shadowing only the IP address 10.0.0.1, and a firewall with its own access list containing only one

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

[2]https://github.com/netgroup-polito/verefoo



F1 Forwarding domain						
#	action	ipSrc	ipDst	portSrc	portDst	trProto
1	Allow	10.0.0.1	30.0.5.1	*	*	*
D	Deny	*	*	*	*	*
$\mathcal{I}_a$		(10.0.0.1, 30.0.5.1, *, *, *)				
$\mathcal{I}_d$		$\neg\mathcal{I}_a$				

NAT 30.0.0.1 Transformation domain	
Sources	10.0.0.1
$\mathcal{D}_1$ (Shadowing)	(10.0.0.1, $\neg$10.0.0.1 $\wedge$ $\neg$30.0.0.1, *, *, *)
$\mathcal{D}_2$ (Reconversion)	($\neg$10.0.0.1 $\wedge$ $\neg$30.0.0.1, 30.0.0.1, *, *, *)
$\mathcal{D}_3$ (Forwarding)	$\neg\mathcal{D}_1 \wedge \neg\mathcal{D}_2$

Fig. 2: Example scenario

rule. In this scenario, the only security policies specified by the user refer to all traffic flows originating from the client sub-network 10.0.0.0/24 and directed to the server 30.0.5.1, without any restriction on port numbers and protocol type. In particular, the flows linking 10.0.0.0/24 and 30.0.5.1 can take two possible paths: one passing through the NAT and one passing through the client sub-network 20.0.0.0/24.

IV. Model

In this section, we present the formal models of the main network elements (packet classes, network topology and functions, network security policies, traffic flows). These models are partially mutated from the modeling approaches proposed in [11] and [13], as they were already defined to be compatible with the two network security automation operations analyzed in this paper, i.e., the verification of connectivity properties, and the refinement of user-specified policies into security function configuration. However, they have been generalized so that they could be applied to both the algorithms for computing the Atomic Flows and Maximal Flows.

A. Packet class model

Definition IV.1. Packet class (traffic): *A packet class, also named traffic, t, is modeled as a predicate defined over the values of the packet fields. In particular, t is a disjunction of predicates, where each predicate is defined over a single field.*

This definition is general enough to support the formalization of packet class models suitable for solving different network security management problems. Depending on the packet fields that must be considered to solve them, new predicates defined over them may be included to the disjunction of predicates modeling t.

In the following, we present a special case of packet class model, where the packet fields on which the predicates of t are defined are the ones of the IP 5-tuple. This specific formulation has been mutated from the two studies proposing the resolution strategies with which we aim to analyze the two flow computation algorithms, i.e., [11] and [13]. In those studies, it has been already proved to achieve higher efficiency for verification and refinement algorithms with respect to state-of-the-art alternatives, hence the decision of using it as a starting point for the approach discussed in this paper.

In this particular case, a packet class t is formally modeled as a disjunction of predicates $q_{t,1} \vee q_{t,2} \vee ... \vee q_{t,n}$, where each $q_{t,i}$ is defined over the 5-tuple fields. A packet belongs to class t if and only if its 5-tuple satisfies at least one $q_{t,i}$. Each $q_{t,i}$ represents the conjunction of five predicates, one for each field of the 5-tuple. For sake of simplicity, each $q_{t,i}$ is written as

$$q_{t,i} = (ipSrc, ipDst, portSrc, portDst, trProto) \quad (1)$$

where *ipSrc*, *ipDst*, *portSrc*, *portDst* and *trProto* are the 5 predicates.

The predicates about source and destination IPv4 addresses *ipSrc* and *ipDst* are conjunctions of four predicates, one for each byte of the address. Each one of the four predicates can represent a single integer value or a range of values, not exceeding the range 0 to 255. The predicates that make up *ipSrc* or *ipDst* are concisely written by means of the dotted-decimal notation $ip_1.ip_2.ip_3.ip_4$. The range [0, 255] is concisely represented by the wildcard $*$. If ip_i is a range, the predicates on its right must be $*$. For example, *ipSrc* = 130.192.5.$*$ stands for the predicate $x_1 = 130 \wedge x_2 = 192 \wedge x_3 = 5$, where x_i is the variable representing the i-th byte of the source IP address packet field, and this predicate identifies all the IP addresses matching 130.192.5.0/24.

The predicates about source and destination ports *portSrc* and *portDst* can identify either a single integer number or a range of values, not exceeding the range 0 to 65535, and the same notation used for each byte of an IP address is also used for the port number, with the range [0, 65535] symbolized by the wildcard $*$. For example, 80 stands for the predicate $x = 80$ and $[80, 100]$ stands for the predicate $x <= 100 \wedge x >= 80$ where x is the variable that represents the port field.

The predicate about the transport-level protocol *trProto* can identify a single value or a subset of values among a finite set of possible values (e.g., a set including the "TCP" and "UDP" values). The set of all the possible values in this set is concisely symbolized by the wildcard $*$.

In all our models, we use the "." notation to denote a specific tuple element (e.g., given a tuple $t = (a, b, c)$, $t.a$ identifies element a of tuple t). Therefore, each sub-predicate of a $q_{t,i}$ predicate can be denoted with this notation for sake of conciseness, e.g., $q_{t,i}.ipSrc$ represents the sub-predicate of the $q_{t,i}$ predicate defined over the source IP address.

Let us denote Q the set of all the predicates $q_{t,i}$ that can be specified with the above notation, and T the set of all the disjunctions of such predicates, i.e., the set of all packet classes t that can be represented by this model. It can be proved that T is closed under conjunction, disjunction and negation. Given

two traffic predicates $t_1, t_2 \in T$, t_1 is said to be a sub-traffic of t_2, written $t_1 \subseteq t_2$, if t_1 represents a subset of the packets represented by t_2, i.e., if $t_1 \Rightarrow t_2$.

B. Network model

The network topology is modeled as a tuple (N, L), where N is the set of the vertices representing network nodes, whereas L is the set of the links representing directed network connections. The nodes represent the network endpoints and functions (i.e., web clients, web servers, routers, firewalls, VPN gateways, load balancers).

Definition IV.2. Network Function (NF): *The behavior of an NF is modeled abstractly by means of two functions, which captures respectively the forwarding behavior (i.e., which input packets are discarded by the functions) and the transformation behavior (i.e., how the packets are modified before being forwarded to the next hop towards their destination).*

The forwarding behavior of a function node $n_i \in N$ is modeled with the predicate $deny_i : T \rightarrow \mathbb{B}$, which maps an input packet class to the Boolean value true, if all the packets of that class are dropped by the function, as a consequence of its configuration. I_i^d denotes the largest class of packets t such that $deny_i(t) =$ true. Instead, the complement of I_i^d, i.e., the class of packets which are not discarded by the function, is denoted as $\mathcal{I}_i^a$. From this definition, it derives that $\mathcal{I}_i^d \vee \mathcal{I}_i^a =$ true and $\mathcal{I}_i^d \wedge \mathcal{I}_i^a =$ false.

The transformation behavior of a node $n_i \in N$ is modeled with the function $\mathcal{T}_i : T \rightarrow T$, which maps an input packet class to another class. For many network functions such as forwarders, traffic monitors and load balancers, $\mathcal{T}$ is the identity function, because they do not apply any transformation to the packets that reach their input ports. In this case, both the $\mathcal{T}$ domain and co-domain correspond to the T set. Instead, other functions show a more complex behavior. An example is a type of *Network Address Translator* (NAT) that can perform only address translation, without the feature of port translation. The configuration of such NAT is commonly characterized by m shadowed IP addresses, i.e., the IP addresses that the NAT translates, represented by the predicates the $p_1, ..., p_m$, and by l public IP addresses, used to replace the shadowed addresses and denoted by the $a_1, ..., a_l$ predicates. With such configuration, this NAT can perform three different transformations on an input packet: 1) the source address is translated into a public address of the NAT (shadowing), if the source address is a shadowed address, while the destination address is not; 2) the destination address is reconverted into a shadowed address (reconversion), if the source address is not a shadowed address and the destination address is a public one; 3) if no previous condition is met, the packet is not modified. For such a function that operates different transformations for different packet classes, the transformer can be expressed as $\mathcal{T}_i(t) = \vee_j(\mathcal{T}_{i,j}(\mathcal{D}_{i,j} \wedge t))$, where $\mathcal{T}_{i,j} : T \rightarrow T$ is the transformer applied for the packet class defined by predicate $\mathcal{D}_{i,j}$. In the case of NAT, we have $\mathcal{T}_i(t) = \mathcal{T}_{i,1}(\mathcal{D}_{i,1} \wedge t) \vee \mathcal{T}_{i,2}(\mathcal{D}_{i,2} \wedge t) \vee \mathcal{T}_{i,3}(\mathcal{D}_{i,3} \wedge t)$, where $\mathcal{T}_{i,1}$ is the shadowing transformer, $\mathcal{T}_{i,2}$ is the reconverting transformer and $\mathcal{T}_{i,3}$ is the identity transformer that is applied in all other cases. Then, considering a generic traffic $t = \vee_{k=1}^{h}(q_{t,k})$, the predicates $\mathcal{D}_{i,j}$ and the transformers $\mathcal{T}_{i,j}$ can be defined as follows.

$$\mathcal{D}_{i,1} = \vee_{x=1}^{m}(p_x, \neg(\vee_{z=1}^{m}(p_z)), *, *, *) \tag{2}$$

$$\mathcal{T}_{i,1}(t) = \vee_{y=1}^{l} \vee_{k=1}^{h} (a_y, q_{t,k}.ipDst, q_{t,k}.portSrc, q_{t,k}.portDst, q_{t,k}.trProto) \tag{3}$$

$$\mathcal{D}_{i,2} = \vee_{y=1}^{l}(\neg(\vee_{x=1}^{m}(p_x)), a_y, *, *, *) \tag{4}$$

$$\mathcal{T}_{i,2}(t) = \vee_{x=1}^{m} \vee_{k=1}^{h} (q_{t,k}.ipSrc, p_x, q_{t,k}.portSrc, q_{t,k}.portDst, q_{t,k}.trProto) \tag{5}$$

$$\mathcal{D}_{i,3} = \neg(\mathcal{D}_{i,1}) \wedge \neg(\mathcal{D}_{i,2}) \tag{6}$$

$$\mathcal{T}_{i,3}(t) = t \tag{7}$$

An example of firewall and NAT models is reported at the bottom of Fig. 2.

C. Traffic flows and network security policies models

Definition IV.3. Traffic flow: *A traffic flow represents how a packet class, generated by a source $n_s \in N$, crosses an ordered list of intermediate network functions before reaching the destination $n_d \in N$, and how these functions may transform the packets belonging to this class before possibly forwarding them to the next hop.*

A traffic flow $f \in F$, where F is the set of all flows, is formally modeled as a list of alternating nodes and predicates, $[n_s, t_{sa}, n_a, t_{ab}, n_b, ..., n_k, t_{kd}, n_d]$. Each node in the list corresponds to a node crossed by the flow in the path, starting from the source node n_s (that generates traffic t_{sa}) and arriving at the destination node n_d (that receives traffic t_{kd}). Each generic traffic t_{ij} is the class of packets transmitted from node n_i to n_j in the flow. While crossing a node, the traffic can be forwarded, possibly changed, or dropped. In this way, traffic flows are used to describe the forwarding and transformation behavior of a network and of its NFs. The main advantage of this approach, compared to the alternative modeling approaches, is that the NFs can be modeled in a simpler way, as the models do not need to deal with all the single packets but they can deal with a few equivalent classes of packets.

A computer network may be crossed by countless different traffic flows. However, in the formalization proposed in this paper, only a limited subset of flows are modeled, i..e, the ones that must be considered when checking if a network security policy is satisfied in a verification problem, or when transforming a policy into the a function configuration in a refinement problem.

Definition IV.4. Connectivity network security policy: *A network security policy expresses security requirements that must be fulfilled in a computer network. Among the possible policies that may be defined, the connectivity ones specify which traffic flows must not (reachability policies), and which must be prohibited from reaching it (isolation policies). In the remainder of this paper, we will name connectivity policies as policies for sake of conciseness.*

A network security policy is modeled as $p = (C, a)$, where a is the action to perform on the network packets that match the condition The condition C is a predicate similar to the ones defined for modeling packet classes. In relation to the specific example of packet class model presented in Subsection IV-A, C is modeled as $C = (ipSrc, ipDst, portSrc, portDst, trProto)$. The predicates *ipSrc* and *portSrc* specify the traffic sources the policy refers to. Instead, the predicates *ipDst*, *portDst*, and *trProto* specify the traffic destinations and the protocol the policy refers to. Instead, the action a may be *allow* in case of a reachability policy, *deny* in case of an isolation policy.

For this specific network security policy model, a flow $f = [e_s, t_{s,a}, ..., t_{k,d}, e_d]$ satisfies C if the following three conditions are satisfied: 1) its source and destination endpoints e_s, e_d have IP addresses matching *ipSrc* and *ipDst* respectively, i.e., $\alpha(e_s) \subseteq C.ipSrc$ and $\alpha(e_d) \subseteq C.ipDst$; 2) its source traffic satisfies *ipSrc* and *portSrc*, i.e., $t_{sa} \subseteq (C.ipSrc, *, C.portSrc, *, *)$; 3) its destination traffic satisfies *ipDst*, *portDst*, and *trProto*, i.e., $t_{kd} \subseteq (*, C.ipDst, *, C.portDst, C.trProto)$. Similar conditions are formulated with minor changes, if the predicates of the packet class models are defined over different packet fields.

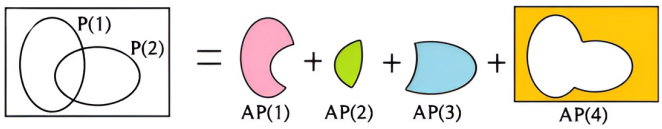
Let then $F_p \subseteq F$ denote the set of flows that satisfy $p.C$. Therefore, it follows that all the subflows of a flow that is in F_p are in F_p too.

The definition of traffic flows given above leaves some freedom concerning the granularity of flows, i.e., it is possible to consider fewer flows characterized by larger packet classes or more flows characterized by simpler packet classes. Therefore, it is suitable for the definition of both the Atomic Flows and Maximal Flows algorithms, as it will be detailed in next sections.

V. Atomic Flows

The first flow computation algorithm that we described is the one that computes Atomic Flows, entities based on the concept of Atomic Predicates. According to this concept, each complex predicate used to model the network can be split into a disjunction of simple and minimal Atomic Predicates, that are unique and completely disjoint. Then, it is possible to collect them in a set, i.e., the set of Atomic Predicates of the network, and replace each original complex predicate as a disjunction of some of the predicates of this set. Being unique, Atomic Predicates can be assigned unique integer identifiers. In this way, each complex predicate can be represented as a set of integers, where each integer is the identifier of each Atomic Predicate composing the disjunction that characterizes the complex predicate. The advantage of this approach is that it allows to use integers in all the computations, instead of using more complex explicit representations for each predicate (e.g., BDDs, Tuple Representations, or Wildcard Expressions, just to mention a few used in the literature).

Fig. 3 shows an example of how complex predicates can be represented as disjunctions of Atomic Predicates. Starting from two complex predicates of the network P(1) and P(2), representing respectively packets traveling from the sub-network 10.0.0.0/24 to 30.0.5.1 and packets traveling from 10.0.0.1 to the sub-network 30.0.5.0/24, the corresponding set of Atomic Predicates is computed. Note that P(1) and P(2) have a non-empty intersection. The four resulting Atomic Predicates are AP(1), AP(2), AP(3), and AP(4), and they are minimal and disjoint (i.e., they do not overlap). In this way, P(1) can be expressed as the disjunction of AP(1) and AP(2), while P2 as the disjunction of AP(2) and AP(3).



P(1) = (10.0.0.*, 30.0.5.1, *, *, *)
P(2) = (10.0.0.1, 30.0.5.*, *, *, *)
AP(1) = (10.0.0.* ∧ ¬10.0.0.1, 30.0.5.1, *, *, *)
AP(2) = (10.0.0.1, 30.0.5.1, *, *, *)
AP(3) = (10.0.0.1, 30.0.5.* ∧ ¬30.0.5.1, *, *, *)
AP(4) = (¬10.0.0.*, *, *, *, *) ∪ (*, ¬30.0.5.*, *, *, *)

Fig. 3: From complex predicates to Atomic Predicates

From the set of Atomic Predicates it is possible to compute the set of Atomic Flows.

Definition V.1. Atomic Flows: *A flow* $f = [n_s, t_{sa}, n_a, t_{ab}, n_b, ..., n_k, t_{kd}, n_d]$ *is defined as atomic if each traffic* t_{ij} *is an Atomic Predicate.*

The goal is to compute flows that are as simple as possible and mutually disjoint because they contain only Atomic Predicates. In this way, Atomic Flows are simple lists of alternating nodes and integers.

A. Atomic Flows algorithms

This subsection describes the two algorithms to compute respectively the set of Atomic Predicates and the corresponding set of Atomic Flows, starting from the models describing the security policies the network must satisfy and the behavior of the network functions.

Algorithm 1 is used to compute the set of Atomic Predicates of the network.

The first objective of Algorithm 1 is the creation of the set $\mathcal{P}$, containing all the predicates that are useful to model both the computer network and the security policies to be enforced in it (lines 1-6). A predicate is included in $\mathcal{P}$ if it is useful to discriminate the packets a security policy refers to or to identify a domain of an NF crossed by at least one of the paths a security policy refers to. More precisely, $\mathcal{P}$ collects three predicate classes: (i) predicates representing the source packets identified by a policy condition (line 3); (ii) predicates representing the destination packets identified by a policy condition (line 4); (iii) predicates describing the forwarding and transformation domains ($\mathcal{I}_i^a, \mathcal{I}_i^d, \mathcal{D}_i$) for each NF, such as firewalls and NATs, belonging to one of the paths associated with a security policy (line 6). Predicates belonging to these three classes are complex and not yet atomic.

Algorithm 1 for computing the Atomic Predicates

Input: a set of n security policies $p = (C, a)$, a set of m intermediate network functions, each one with its $\{\mathcal{I}^a, \mathcal{I}^d\}$, $\{\mathcal{D}\}$
Output: the set of Atomic Predicates $\mathcal{B}$

1: $\mathcal{P} \leftarrow \{\text{false}\}$
2: **for** $i = 0, 1, ..., n$ **do**
3: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{(C_i.\text{ipSrc}, *, C_i.\text{portSrc}, *, *)\}$
4: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{(*, C_i.\text{ipDst}, *, C_i.\text{portDst}, C_i.\text{trProto})\}$
5: **for** $i = 0, 1, ..., m$ **do**
6: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_i^a, \mathcal{I}_i^d\} \cup \{\mathcal{D}_i\}$
7: $\mathcal{B} \leftarrow \mathcal{A}(\mathcal{P})$
8: $\mathcal{R} \leftarrow \{\text{false}\}$
9: **for** $i = 0, 1, ..., m$ **do**
10: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{\mathcal{T}_i(b) |$ for each $b \in \mathcal{B}\}$
11: **if** $\mathcal{B} = \mathcal{A}(\mathcal{P} \cup \mathcal{R})$ **then**
12: $\quad$ **return** $\mathcal{B}$
13: **else**
14: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{R}$, $\mathcal{B} \leftarrow \mathcal{A}(\mathcal{P} \cup \mathcal{R})$
15: $\quad$ **goto** line 9

Algorithm 2 for computing the atomic flows

Input: one security policy p, one path with endpoints e_s and e_d and middleboxes $\mathcal{L}$=[n_1,n_2,...,n_m], the set $\mathcal{B}$ of Atomic Predicates computed by Algorithm 1
Output: a set of Atomic Flows F_a

1: $F_a \leftarrow \emptyset$
2: $\mathcal{B}_0 \leftarrow \{b_1, b_2, ..., b_{m_t}\}$ $\forall b_i : b_i.\text{ipSrc} \wedge p.\text{ipSrc} \neq \emptyset$ and
3: $\quad b_i.\text{portSrc} \wedge p.\text{portSrc} \neq \emptyset$ and $b_i \in \mathcal{B}$
4: **for** $b \in \mathcal{B}_0$ **do**
5: $\quad$ **for** $f \in$ RECURSIVEGEN$(1, b)$ **do**
6: $\quad\quad f_a \leftarrow [e_s, b] + f$
7: $\quad\quad F_a \leftarrow F_a \cup \{f_a\}$
8: **return** F_a

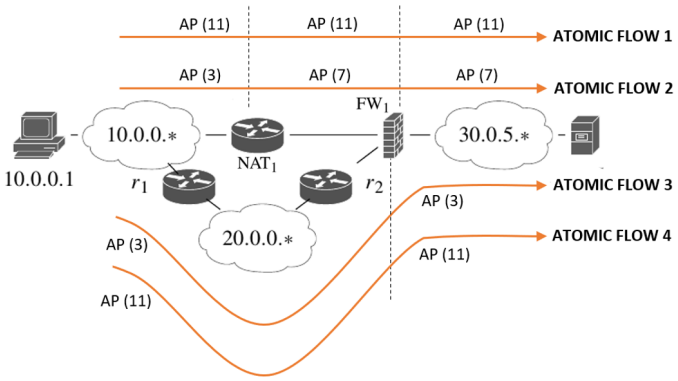
9: **function** RECURSIVEGEN(i, b)
10: $\quad$ **if** $i == m + 1$ **then**
11: $\quad\quad$ **if** $b.\text{ipDst} == \alpha(e_d)$ and $b.\text{portDst} \wedge p.\text{portDst} \neq \emptyset$ and
12: $\quad\quad\quad b.\text{trProto} \wedge p.\text{trProto} \neq \emptyset$ **then**
13: $\quad\quad\quad$ **return** $\{[e_d]\}$
14: $\quad\quad$ **else return** $\emptyset$
15: $\quad$ **if** $b.\text{ipDst} == \alpha(n_i)$ **then return** $\emptyset$
16: $\quad r \leftarrow \mathcal{T}_i(b)$
17: $\quad \mathcal{B}_t \leftarrow \{b_1, b_2, .., b_{m_t}\}$ such that $\vee_{j=1}^{m_t} b_j = r$ and $b_j \in \mathcal{B}_i$
18: $\quad F_t \leftarrow \emptyset$
19: $\quad$ **for** $b_t \in \mathcal{B}_t$ **do**
20: $\quad\quad$ **for** $f \in$ RECURSIVEGEN$(i + 1, b_t)$ **do**
21: $\quad\quad\quad f_t \leftarrow [n_i, b_t] + f$
22: $\quad\quad\quad F_t \leftarrow F_t \cup \{f_t\}$
23: $\quad$ **return** F_t

The second objective of Algorithm 1 is the actual computation of the Atomic Predicates (lines 7-15). In particular, the algorithm transforms the set $\mathcal{P}$ into the corresponding set $\mathcal{B}$ of Atomic Predicates, unique for the entire network, applying the function $\mathcal{A}$ (line 7). $\mathcal{A}$ is a standard function that allows, given a set of predicates, to split them according to their mutual intersections. In literature, there are several algorithms to perform this task. An example can be found in [10]. However, $\mathcal{P}$ only contains the input packet classes of the network functions and not the transformed ones, so the related set $\mathcal{P}$ does not include Atomic Predicates related to complex predicates output by transformers. In order to compensate for this, Algorithm 1 applies all the possible transformations $\mathcal{T}_i$ to each Atomic Predicate of the set $\mathcal{B}$ that matches a specific input class of those transformers, obtaining in this way the transformed predicates (line 10). The transformed predicates are included into the set $\mathcal{R}$, and the function $\mathcal{A}$ is applied to the union of the sets $\mathcal{B}$ and $\mathcal{R}$, producing a new set $\mathcal{B}$ that replaces the previous one (line 11). All these operations from line 9 to line 11 are iterated, until the new set $\mathcal{B}$ does not undergo any new change. This means that all the possible combinations made of a input predicate and corresponding transformed output predicate have been considered, and $\mathcal{B}$ has thus become the set of Atomic Predicates representative for both input predicates and transformed ones (line 12).

The worst-case time complexity of Algorithm 1 can be estimated as the sum of the time complexities of four sequential code blocks. Lines 1-4 have $O(n)$ complexity, because $O(1)$ operations are performed on each one of the n input security policies. Lines 5-6 have $O(m)$ complexity, because $O(1)$ operations are performed on each one of the m input middleboxes. Line 7 has $O(\mathcal{A})$ complexity, where $\mathcal{A}$ is the external function that is called to compute the atomic predicates of a given set of input predicates. Referring to the example of $\mathcal{A}$ presented in [10], this function is linear in the number of input predicates. Lines 8-5 have $O(m) \cdot (O(m) + O(\mathcal{A}))$ complexity because that code block consists of two nested loops. The external loop, represented by the *goto* directive in the algorithm pseudocode, iterates m times in the worst case. This is the case where all m middleboxes are arranged in a single sequential chain, and the output of each $\mathcal{T}_i$ transformation function may take up to m steps to impact the behavior of the same function related to the last node in the sequence. Then, in this external loop, lines 9-10 represent an internal loop iterating on the m middleboxes, and sequentially the function $\mathcal{A}$ is executed at line 1. In view of this analysis, ignoring the complexity of the second and third code blocks as they are dominated by the one of the fourth block, the overall worst-case time complexity of Algorithm 1 is $O(n) + O(\mathcal{A}) \cdot O(m) + O(m^2)$. Among these three terms, it is not possible to establish theoretically that one dominates the others, because it depends on the different scenarios to which the algorithm is applied (e.g., in some scenarios there may be more policies than middleboxes, or vice versa).

The result of Algorithm 1, representing the set of Atomic Predicates, is then used as input for Algorithm 2, which computes the Atomic Flows.

Algorithm 2 considers one security policy $p = (C, a)$ and one path, characterized by endpoints e_s and e_d and a list of middleboxes $\mathcal{L} = [n_1, n_2, ..., n_m]$, at a time. Nodes e_s and e_d are endpoints whose IP addresses match respectively C.ipSrc and C.ipDst, as explained in Subsection IV-C. At the beginning of Algorithm 2, the traffic generated by e_s (source node of the policy) is grouped in a subset $\mathcal{B}_0$, which represents



Atomic predicates with their integer identifier	
AP(1) = (10.0.0.1, 10.0.0.1, *, *, *)	AP(2) = (10.0.0.1, 30.0.0.1, *, *, *)
AP(3) = (10.0.0.1, 30.0.5.1, *, *, *) *, *, *)	AP(4) = (10.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1 ∧ ¬30.0.5.1, *, *, *)
AP(5) = (30.0.0.1, 10.0.0.1, *, *, *)	AP(6) = (30.0.0.1, 30.0.0.1, *, *, *)
AP(7) = (30.0.0.1, 30.0.5.1, *, *, *)	AP(8) = (30.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1 ∧ ¬30.0.5.1, *, *, *)
AP(9) = (10.0.0.* ∧ ¬10.0.0.1, 10.0.0.1, *, *, *)	AP(10) = (10.0.0.*∧ ¬10.0.0.1, 30.0.0.1, *, *, *)
AP(11) = (10.0.0.*∧ ¬10.0.0.1, 30.0.5.1, *, *, *)	AP(12) = (10.0.0.*∧ ¬10.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1 ∧ ¬30.0.5.1, *, *, *)
AP(13) = (¬10.0.0.*∧ ¬30.0.0.1, 10.0.0.1, *, *, *)	AP(14) = (¬10.0.0.*∧ ¬30.0.0.1, 30.0.0.1, *, *, *)
AP(15) = (¬10.0.0.*∧ ¬30.0.0.1, 30.0.5.1, *, *, *)	AP(16) = (¬10.0.0.*∧ ¬30.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1 ∧ ¬30.0.5.1, *, *,*)

Transformation for NAT1	
$\mathcal{D}_1$ (Shadowing)	(3) becomes (7), (4) becomes (8)
$\mathcal{D}_2$ (Reconversion)	(10) becomes (9), (14) becomes (13)
$\mathcal{D}_3$ (Forwarding)	(1), (5), (11), (12), (15), (16) are simply forwarded
	(2), (6) reach their destination in the NAT

Forwarding for FW1	
$\mathcal{I}^a$	(3) allowed to pass
$\mathcal{I}^d$	(1), (2), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16) dropped

Fig. 4: Atomic Flows

the disjunction of all the Atomic Predicates of $\mathcal{B}$ whose ipSrc and portSrc are equal to the ones expressed by the condition of the policy (line 3). Then, starting from the predicates in $\mathcal{B}_0$, the algorithm computes recursively all related Atomic Flows, which represent the evolution of each single predicate along the specific path (line 5). In greater detail, each single Atomic Predicate b, belonging to $\mathcal{B}_0$, is propagated along the path that links source to destination, taking in consideration the fact that, crossing a node, it can be transformed into one or more different disjoint Atomic Predicates (lines 16-17). At each recursion level, some pruning is also performed in order to discard the flows that are not part of the solution. For example, flows that arrive at the destination with an Atomic Predicate whose ipDst, portDst and trProto do not match the corresponding fields of the policy condition (line 14) or that have already reached their destination before reaching the destination of the path (i.e., when the IP address of an intermediate node matches with the ipDst field of the actual Atomic Predicate) (line 15) are removed.

The worst-case time complexity of Algorithm 2 can be estimated as the sum of the time complexities of two sequential code blocks. Lines 1-3 have $O(|\mathcal{B}|)$ complexity, because $O(1)$ operations are performed on each Atomic Predicates of the $\mathcal{B}$ set. Then, lines 4-8 call iteratively a recursive function described in lines 10-23. On the one hand, the loop has $O(|\mathcal{B}|)$ complexity, because it iterates at most $|\mathcal{B}|$ times in the worst case where all elements of $\mathcal{B}$ are included in $\mathcal{B}_0$. On the other hand, the recursive function has $O(|\mathcal{B}|^m)$ complexity. The reason is that at each level of recursion, the function makes $|\mathcal{B}|$ recursive calls, while the depth of recursion is determined by the parameter i, which starts at 1 and goes up to m. All other operations in the recursive function simply have $O(1)$ complexity. In conclusion, the complexity of the second block is $O(|\mathcal{B}|^{m+1})$ and, as it dominates the complexity term of the first block, it coincides with the worst-case time complexity of the overall Algorithm 2.

B. Atomic Flows example

This subsection illustrates how the algorithms proposed for the Atomic Flow groping strategy works, when they are executed to identify all the traffic flows related to an input security policy p, with condition $p.C = (10.0.0.*, 30.0.5.1, *, *, *)$ and action $p.a = deny$, in the network topology introduced in the running example of Section III. The discussion of this example will be supported by the information provided in Fig. 4, which will be progressively described in the following.

First, Algorithm 1 is executed to compute the set of Atomic Predicates $\mathcal{B}$ related to the input policy p and the NFs of the network topology. As a preliminary operation, it computes the set $\mathcal{P}$, including:

- the predicate describing the source traffic (10.0.0. *, *, *, *, *), representing all the possible packets generated in the sub-network 10.0.0.0/24, on the left of Fig. 4 (line 3 of Algorithm 1);
- the predicate describing the destination traffic (*, 30.0.5.1, *, *, *), representing all the possible traffic that can arrive as input to the server on the right of Fig. 4, without any restriction on port number and protocol type (line 4 of Algorithm 1);
- the predicates related to the input and transformation domains of the NFs found along each of the two possible paths (line 6 of Algorithm 1). The NFs crossed by at least one path are the NAT and the firewall. Their corresponding input domains can be seen in the two inner tables of Fig. 2: predicates $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}_3$ for the NAT and predicates I_a and I_d for the firewall.

This initial set of predicates $\mathcal{P}$ is thus used by Algorithm 1 to compute the corresponding set of Atomic Predicates $\mathcal{B}$ through the function $\mathcal{A}$ (line 7 of Algorithm 1). However, the set $\mathcal{B}$ thus computed does not include Atomic Predicates representing the results of possible transformations that can occur to a packet while crossing the network, but only represent the predicates of $\mathcal{P}$.

Next, Algorithm 1 iterates over each predicate in $\mathcal{B}$ and checks if it intersects any input transformation domain of an NF found along a path (i.e., the NAT in this example). If an intersection is found with a domain $\mathcal{D}_i$, then the corresponding transformation $\mathcal{T}_i$ is applied (line 10 of Algorithm 1). All the computed transformed predicates are thus included into a temporary set $\mathcal{R}$ to get the resulting transformed predicate. For example, the Atomic Predicate AP(1) = (10.0.0.1, 30.0.5.1, *, *, *), shown in Fig. 4, intersects with $\mathcal{D}_1$ = (10.0.0.1, $\neg$10.0.0.1 $\wedge$ 30.0.0.1, *, *, *) of the NAT, so the corresponding Shadowing operation is applied to it. The resulting transformed predicate (30.0.0.1, 30.0.5.1, *, *, *) is then inserted into $\mathcal{R}$.

After computing the set $\mathcal{R}$, Algorithm 1 adds its element to $\mathcal{P}$, and re-applies the function $\mathcal{A}$ to it. If the new set of Atomic Predicates is the same as the one previously computed, the algorithm ends, as that means the effect of all transformations have been taken into account (lines 11-12 of Algorithm 1). Otherwise, the intersection between the Atomic Predicates and the transformation domains is repeated until the recomputed set $\mathcal{B}$ does not change. In this example, as there is only one function that applies transformation, a single iteration is enough, and the set $\mathcal{B}$ includes all the Atomic Predicates shown in the top-level table of Fig. 4. In this table, each Atomic Predicate is assigned with an integer identifier, since they are mutually disjoint. In this way, all forwarding and transformation domains of the NFs can be rewritten by using these integers. The corresponding models of the NAT and Firewall are shown in the medium-level and bottom-level tables of Fig. 4. In particular, the medium-level table shows how each integer representing a possible input packet class is mapped into an integer representing a transformed packet class for the shadowing and reconversion operations, and it lists the integers representing the packet classes that are not transformed but simply forwarded as they are. Instead, the bottom-level table lists the integers representing the packet lasses the firewall drops, and the ones representing the packet classes it allows to pass.

Starting from the set of Atomic Predicates, Algorithm 2 computes the corresponding set of Atomic Flows. Taking one security policy and one path at a time, the algorithm collects into $\mathcal{B}_0$ all the Atomic Predicates matching with the source part of the security policy condition , i.e., all the Atomic Predicates intersecting (10.0.0. *, *, *, *, *) (line 3 of Algorithm 2). In this example, $\mathcal{B}_0$ includes all the Atomic Predicates from AP(1) to AP(4) and from AP(9) to AP(12), because their sub-predicates about source IP address and source port intersect the ones of the policy (C.ipSrc and C.portSrc), and thus they represent all possible packet classes that the sub-network 10.0.0.0/24 may generate. Note that, in this specific case, there is no restriction about source port numbers, as the policy condition C.portSrc is defined as $*$.

Then, through a recursive function (lines 11-12 of Algorithm 2), all Atomic Flows related to the policy p are computed, by propagating each predicate belonging to $\mathcal{B}_0$ along all possible paths. In this propagation of the predicates representing packet classes, they may intersect with the input forwarding or transformation domain of a crossed NF. In that case, the packet class represented by the predicate may be dropped, or transformed into one or more classes represented by different predicates. An Atomic Flow is created only if, starting from the source, the possibly transformed packet class can reach its destination, i.e., 30.0.5.1. Here, we discuss two exemplifying cases of such propagation, with different outcomes:

- AP(2) = (10.0.0.1, 30.0.0.1, *, *, *) is never able to reach the destination 30.0.5.1, because there is no intersection between the destination IP address sub-predicates of the two predicates (i.e., of AP(2) and the policy condition predicate). Instead, the destination of all Atomic Flows starting with AP(2) as first packet class of their model is the NAT 30.0.0.1. Therefore, the recursive function of Algorithm 2 discards all these Atomic Flows, as they are not valid solutions.
- AP(3) = (10.0.0.1, 30.0.5.1, *, *, *) can reach the destination 30.0.5.1 through both the upper and lower paths of the topology depicted in Fig. 4, so two Atomic Flows are generated. Crossing the lower path, the packet class represented by AP(3) does not undergo any transformation, so AP(3) will be in between any pair of nodes included in the alternating list modeling the flow. Instead, crossing the upper path, the packet class represented by AP(3) is transformed by the NAT into the packet class represented by AP(11), which is the one actually reaching the destination.

At the end of this example, four Atomic Flows are computed, as shown in Fig. 4. Two of them originate from AP(3), while the remaining two originate from AP(11).

VI. Maximal Flows

The second flow computation algorithm that we present is the one that computes Maximal Flows, and it is based on a concept opposite to the idea behind Atomic Flows. In fact, this algorithm aggregates different flows together so as to minimize their number, instead of splitting them into Atomic Flows. In particular, it reduces the number of generated flows by considering only a subset of them, i.e., the set of Maximal Flows. In order to do so, all the flows that behave in the same way while crossing the network are grouped in the same Maximal Flow. These flows can be indicated as subflows of the Maximal Flow which they belong to. Then, for the execution of the resolution algorithms of security management problems, it is enough to consider only the Maximal Flows and not each single flow that they represent.

Definition VI.1. Maximal Flows: *Called F_p the set of all possible flows of the network, the corresponding set of Maximal Flows F_p^M matches the following definition:*

$$F_p^M = \{f_p^M \in F_p | \nexists f \in F_p . (f \neq f_p^M \wedge f_p^M \subseteq f)\}$$

The set F_p^M is defined as a subset of F_p that contains only the flows that are not subflows of any other flow in F_p. All the flows behaving in the same way are aggregated in the same Maximal Flow, and then only Maximal Flows in F_p^M, which has a smaller size than F_p, are considered for the analysis.

Predicates defined within a Maximal Flow are the disjunction of several IP 5-tuples that have been aggregated together. For this reason, they cannot be considered atomic and replaced by integer identifiers.

A. Maximal Flows algorithm

Algorithm 3 is formalized for the computation of the Maximal Flows.

As for Algorithm 2, one security policy and one possible path are considered at a time. For each path, Algorithm 3 computes the related Maximal Flows in an iterative way (line 2). At each iteration, two sets of lists of alternating nodes and predicates, F and F', are computed. The first set F initially contains only the list $[n_0, t_1, n_1, \text{true}\ldots, \text{true}, n_{m+1}]$ (line 3). In this list, t_1 is equal to the predicate (C.ipSrc, *, C.portSrc, *, *), representing the largest traffic that satisfies the source component of the network policy condition we are considering, while all the other traffic entities inside the list are set to true (i.e., the class of all packets). The basic idea of this algorithm is to start with flows that are as large as possible (i.e., which include as many subflows as possible) and then divide the flows into smaller flows only when necessary. For example, a division may be needed when the flow encounters a node in the path in relation to which at least two of its subflows have different behavior.

At each iteration, a forward traversal and a backward traversal on the path p are performed.

In the forward traversal (lines 4-7), each list in F is progressively updated to take into account the way the traffic is transformed by each network function. Each predicate b_i, in input to the node n_i, is intersected with the forwarding domains $\mathcal{I}_i^a$ and $\mathcal{I}_i^d$ (line 5), and the transformation domain D_i of the node (line 6). The corresponding function T is also applied to each traffic that matches the domain D_i (line 7). With this procedure, b_i is thus split into the largest homogeneous subclasses of packets resulting from all the computed intersections. In this way, for each partition of the predicate b_i, a new list is generated and becomes a new Maximal Flow. Then, when the traffic arrives to the destination node, it is restricted with the predicate representing the destination components of the policy we are considering, i.e., (*, C.ipDst, *, C.portDst, C.trProto) (line 8). Note that, in all lines of Algorithm 3, the operator + is used to concatenate multiple lists into a single list. These lists are appended in the order in which they are specified as input to this operator.

A backward traversal is then executed, so as to compute a new set of lists F' (lines 8-10). F' is initialized to contain each element of F, with its last traffic restricted to the largest traffic that satisfy the destination components of the policy, as previously described (line 8). In this way, with the backward traversal, each predicate representing the ingress traffic of a node is changed as the new information about the destination is added (lines 9-10).

After the backward traversal, a new forward traversal starts, to evaluate if the new version of the flow further splits (lines 11-13). The procedure stops when, after the last iteration, the flows in F and F' are the same. This means that both last two traversals have no longer changed the flow (lines 14-15).

For what concerns the worst-case time complexity, Algorithm 3 is characterized by three nested loops. The most external loop starts at line 2 and iterates on all possible paths that can be identified between any pair of endpoints in the input network topology (N, L). Denoting the number of all possible paths as π, the complexity of the most external loop is $O(\pi)$. The central loop is represented by the *goto* directive of line 13 and iterates m times in the worst case, where m is the number of intermediate network functions in the input topology, so it has a $O(m)$ complexity. The reason is similar to the one discussed for a similar loop in Algorithm 1, i.e., the output of each $\mathcal{T}_i$ transformation function may take up to m steps to have an impact on the behavior of the same function related to the last node in the forward traversal, and the same applies to the inverse function $\mathcal{T}_i^{-1}$ function in the backward traversal. Then, internally there are two sequential loops at lines 4-7 and lines 9-10, each one with $O(m)$ complexity as both iterate $O(1)$ operations m times. In summary, multiplying the complexity factor of each nested loop, the overall worst-case time complexity of Algorithm 3 is $O(\pi) \cdot O(m^2)$.

Algorithm 3 computation of F_p^M

Input: a policy $p = (C, a)$, the network topology (N, L)
Output: F_p^M

1: $F_p^M = \emptyset$
2: **for each** path = $[n_0, n_1, ..., n_{m+1}]$ **do**
3: $F \leftarrow \{[n_0, t_1^t, n_1, \text{true}, n_2, ..., \text{true}, n_{m+1}]\}$
4: **for** $i = 1, 2, ..., m$ **do**
5: $F \leftarrow \{l + [b_i \wedge b_i', n_i] + l' \mid l + [b_i, n_i] + l' \in F, b_i' \in \{\mathcal{I}_i^a, \mathcal{I}_i^d\}\}$
6: $F \leftarrow \{l + [b_i \wedge b_i', n_i] + l' \mid l + [b_i, n_i] + l' \in F, b_i' \in \{\mathcal{D}_i\}\}$
7: $F \leftarrow \{l + [b_i, n_i, b_{i+1} \wedge \mathcal{T}_i(b_i), n_{i+1}] + l' \mid l + [b_i, n_i, b_{i+1}, n_{i+1}] + l' \in F\}$
8: $F' \leftarrow \{l + [t_{m+1}^r \wedge b_{m+1}, n_{m+1}] \mid l + [b_{m+1}, n_{m+1}] \in F\}$
9: **for** $i = m, m-1, ..., 1$ **do**
10: $F' \leftarrow \{l + [b_i \wedge \mathcal{T}_i^{-1}(b_{i+1}), n_i, b_{i+1}] + l' \mid l + [b_i, n_i, b_{i+1}] + l' \in F'\}$
11: **if** $F \neq F'$ **then**
12: $F \leftarrow F'$
13: **goto** line 4
14: $F_p^M \leftarrow F_p^M \cup F$
15: **return** F_p^M

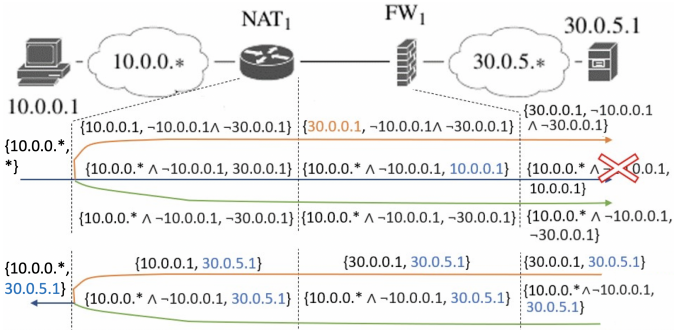


Fig. 5: Maximal Flows Algorithm: first forward and backward traversal

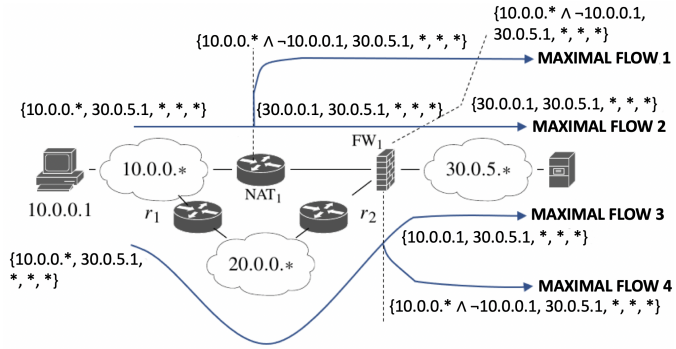


Fig. 6: Maximal Flows

B. Maximal Flows example

This subsection illustrates how the algorithm proposed for the Maximal Flow groping strategy works, when it is executed to identify all the traffic flows related to an input security policy p, with condition $p.C = (10.0.0.*, 30.0.5.1, *, *, *)$ and action $p.a = deny$, in the network topology introduced in the running example of Section III. The discussion of this example will be supported by the information provided in Fig. 5 and Fig. 6.

Specifically, Fig. 5 graphically represents how Algorithm 3 generates two Maximal Flows through the path that includes the NAT, starting from the packet class identified by the predicate representing the source part of the security policy condition, i.e., (10.0.0.*, *, *, *, *). The figure, which only reports source IP and destination IP addresses in the predicates for the sake of conciseness, shows how the initial flow that starts from that initial packet class is progressively divided into smaller flows through the procedures of forward traversal (first line of the figure, below the function chain), and backward traversal (second line of the figure, below the function chain).

In the forward traversal, the initial flow is split into three sub-flows when reaching the NAT, because the predicate (10.0.0.*, *, *, *, *) intersects with all the three input domains of that NF: $\mathcal{D}_1$ =(10.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1, *, *, *), $\mathcal{D}_2$ =(¬10.0.0.1 ∧ ¬30.0.0.1, 30.0.0.1, *, *, *), and $\mathcal{D}_3$ = $\neg\mathcal{D}_1 \wedge \neg\mathcal{D}_2$. In greater detail, the intersection with $\mathcal{D}_1$ generates (10.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1, *, *, *), the one with $\mathcal{D}_2$ generates (10.0.0.* ∧ ¬10.0.0.1, 30.0.0.1, *, *, *), and finally the one with $\mathcal{D}_3$ generates (10.0.0.* ∧ ¬10.0.0.1, ¬30.0.0.1, *, *, *). However, the first two predicates must also undergo a transformation, respectively Shadowing and Reconversion. Therefore, their corresponding output predicates are (30.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1, *, *, *) due to the change to the source IP address, and (10.0.0.* ∧ ¬10.0.0.1, 10.0.0.1, *, *, *) due to the change to the destination IP address.

In the continuation of the forward traversal, the three flows that have been generated so far are not split when reaching the firewall, as all the three predicates representing the packet classes output by the NAT only intersect with the domain $\mathcal{I}^d$ of the firewall, and there is no intersection with $\mathcal{I}^d$. The forward traversal thus ends with the three flows reaching the destination 30.0.5.1. Nevertheless, not all of them are admitted to the next step of the algorithm, which is the backward traversal. Their admission to it is granted only if the predicates representing the packet classes that reached 30.0.5.1 intersect with the destination component of the security policy condition, i.e., (*, 30.0.5.1, *, *, *). The flow whose last packet class predicate is (10.0.0.* ∧ ¬10.0.0.1, 10.0.0.1, *, *, *) is discarded, since its intersection with the destination component of the policy is the empty set.

In the backward traversal, the two admitted flows are propagated from the destination to the original source. The same operations as the ones previously described are repeated, by checking if there is any intersection between packet class predicates and input domains of the NFs. In this example, no other transformations are applied in the backward traversal, and both flows are considered valid when reaching the original source.

At this stage, the algorithm must repeat a second forward and backward traversal. It thus confirms that the two flows have not undergone any other modification or division in this repetition, and therefore they are valid Maximal Flows related to the input security policy.

The same procedure is applied to the other path as well. The only difference is that, in this case, the initial flow is split when reaching the firewall. The intersection with $\mathcal{I}^a$ gives (10.0.0.1, 30.0.5.1, *, *, *), while the intersection with $\mathcal{I}^d$ gives a different predicate.

Finally, Fig. 6 graphically shows that four Maximal Flows are successfully computed with Algorithm 3. Two of them cross the upper path of the depicted network topology, originating from an initial flow that is divided into sub-entities when reaching the NAT. Instead, the other two cross the lower path of the depicted network topology, originating from an initial flow that is divided into sub-entities when reaching the firewall.

VII. Discussion and Experimental Comparison

Depending on the selected flow computation algorithm and on the specific network security management problem to be solved, the comprehensive workflow discussed in Section III is expected to have different execution time. The reason is that the algorithms for computing Atomic and Maximal

Flows have different features. On the one hand, with Atomic Flows the aim is to obtain minimal and disjoint flows, with predicates represented by integers. This simple representation of predicates promises to bring performance advantages, but it also introduces overhead for computing the Atomic Predicates and the related Atomic Flows. On the other hand, with Maximal Flows, the aim is to obtain flows that are indeed more complex, because they aggregate different predicates together, but certainly fewer in number. Besides, for this second algorithm, no initial phase is required to compute Atomic Predicates.

The goal of this section is to evaluate the feasibility and performance of the two traffic flow computation algorithms as stand-alone algorithms and when applied to solve the security policy verification and refinement tasks, so as to underline the advantages and limitations of each one of them. Consequently, two main classes of validation tests have been executed. On the one hand, we have assessed the performance of the flow computation algorithms, independently from the security management problem for which they are used (Subsection VII-A). These tests aim to understand how the execution time varies depending on parameters such as the number of policies, transformer functions like NATs, filtering functions like firewalls, and network endpoint. For the execution of these tests, we have implemented the Atomic Flow algorithm with the Java programming language, as that language was already used for the implemented of the Maximal Flow algorithm in the Verigraph2.0 [11] and Verefoo [13] frameworks. On the other hand, we have analyzed how the Atomic and Maximal Flow formalizations differently affect the resolution of the security policy verification and refinement problems (Subsection VII-B). These tests aim to assess how significantly the choice of flow modeling between the two identified algorithms affects the time performance and memory usage of the overall approach, and to establish which flow computation algorithm is more suitable for each specific analyzed security management problem. For the execution of these tests, we have extended the implementation of the existing Verigraph2.0 and Verefoo frameworks, developing variants that could work with the implemented Atomic Flow algorithm.

The experimental setup used for the validation consists in a machine with an Intel i7-6700 CPU running at 3.40 GHz and 32GB of RAM.

A. Validation for traffic flow computation

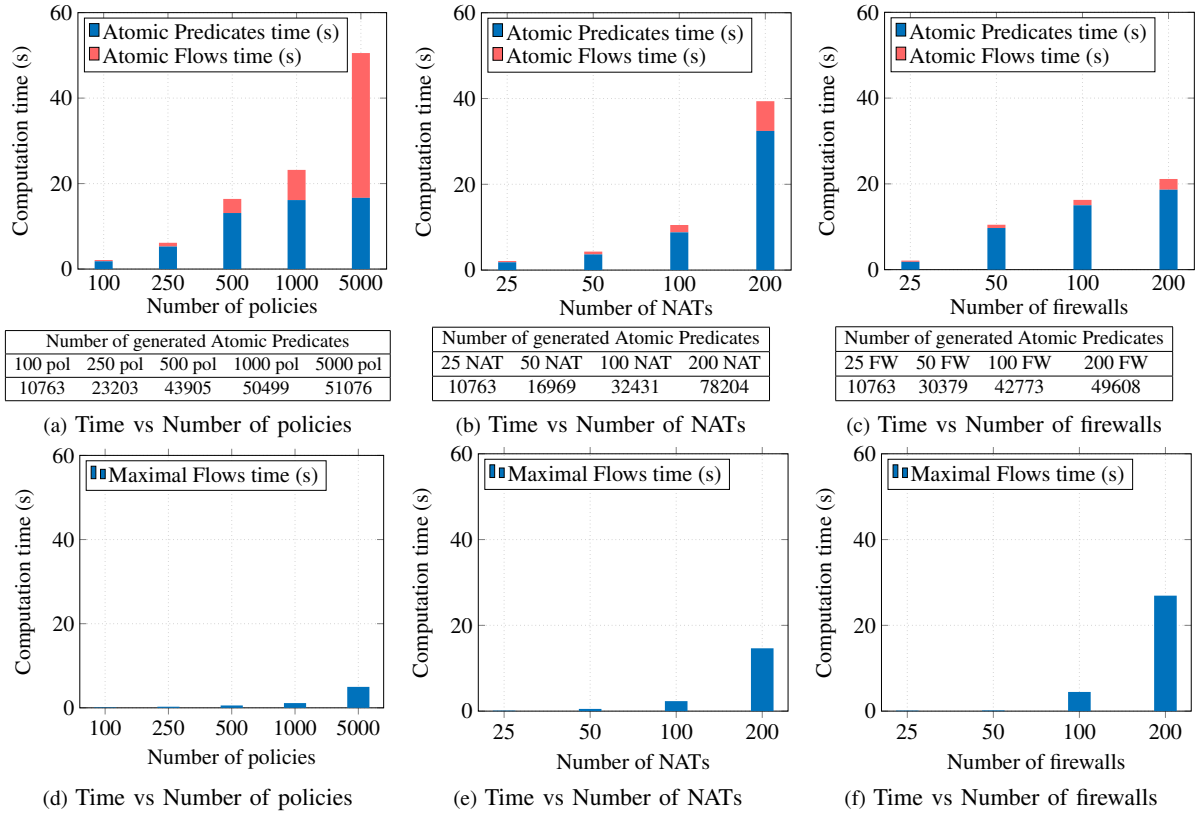
With the first class of tests, we have evaluated the time that the two algorithms take to compute the set of corresponding traffic flows, Atomic or Maximal, and how this time changes as the network varies in size and in a number of parameters. In this evaluation, the networks used as test cases are randomly generated, based on a set of configurable parameters: number of security policies to be satisfied, number of endpoints in the network (inclusive of application clients and servers), number of NATs, and number of firewalls. Then, we followed two strategies for text execution:

- In the first strategy, a preliminary task consists in finding a starting value for each configurable parameter. Once this configuration, named “basic configuration”, has been obtained, we proceed to execute the various tests increasing one parameter at a time, keeping all the others at their basic value. This approach is mainly used to understand which parameters have a greater influence on the total execution time. The chosen "basic configuration" is the one with 100 policies, 200 endpoints, 25 NATs and 25 firewalls.
- In the second strategy, we progressively increase the value of all parameters at the same time in a proportional way, simulating a progressive enlargement of the network.

Fig. 7 shows the results of the tests executed by following the first strategy for both Atomic and Maximal Flow computation algorithms.

Figs. 7a, 7b and 7c depict how the execution time of the Atomic Flow computation algorithm varies when the number of policies, NATs and firewalls respectively increases. As it can be seen, the total time that is required by its execution is the sum of two times, i.e., the time to generate the Atomic Predicates (i.e., Algorithm 1) and the time to generate the Atomic Flows (i.e., Algorithm 2). Below each bar chart, these figures have a table reporting the number of Atomic Predicates generated in each analyzed case. Indeed, this is an important metric which directly affects the total time, since the more the predicates, the more the generated flows.

These figures allowed us to analyze the impact of each parameter of the network configuration for the performance of this algorithm. (i) *Number of policies* (Fig. 7a): As described in Algorithm 1, a predicate is generated for each source and for each destination of a policy. So, the more the policies, the more the starting predicates, and, consequently, the resulting Atomic Predicates. Consequently, the time to compute Atomic Predicates increases because there are more predicates to convert. The time to compute the Atomic Flows increases as well, because Algorithm 2 needs to compute the flows related to a larger set of policies. Regarding the number of generated Atomic Predicates, in the last case, its value saturates to 51076, because there is an Atomic Predicate for each possible pair of source and destination endpoints. (ii) *Number of NATs* (Fig. 7b): This is the most affecting parameter in the number of generated Atomic Predicates. Since NATs introduce transformations, a higher number of predicates to represent the input classes (transformation domain) and the results after the transformation are necessary. Both the considered times increase, since there are more Atomic Predicates to manage. (iii) *Number of firewalls* (Fig. 7c): Here, the reasoning is similar to the one done for NATs. For each firewall, the predicates representing its forwarding domain are added to the set of predicates to be converted into Atomic ones. In this case, the impact is less significant since firewalls are not transformers, so the predicate is either blocked or forwarded, but never changed. Therefore, no new predicates are added as a result of the transformation. Moreover, a general consideration can be drawn with respect to the number of application clients



Number of generated Atomic Predicates				
100 pol	250 pol	500 pol	1000 pol	5000 pol
10763	23203	43905	50499	51076

(a) Time vs Number of policies

Number of generated Atomic Predicates			
25 NAT	50 NAT	100 NAT	200 NAT
10763	16969	32431	78204

(b) Time vs Number of NATs

Number of generated Atomic Predicates			
25 FW	50 FW	100 FW	200 FW
10763	30379	42773	49608

(c) Time vs Number of firewalls

(d) Time vs Number of policies

(e) Time vs Number of NATs

(f) Time vs Number of firewalls

Fig. 7: Tests on Atomic and Maximal Flows algorithms (first strategy)

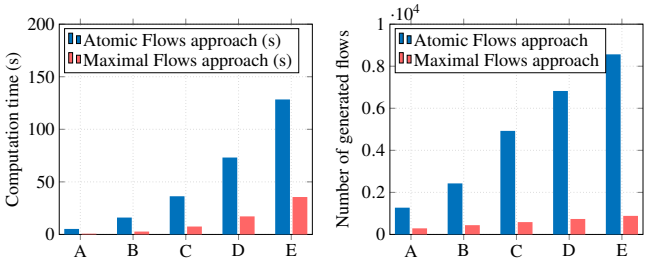
and servers in the network topology. This parameter does not influence the total time required by the algorithm. In general, endpoints do not affect the number of generated Atomic Predicates because such predicates are computed starting from the policies and not for each endpoint of the network. Indeed, there is a starting predicate for every pair of source and destination nodes of a policy. Therefore, if an endpoint is not included in any policy, it is not considered in the set of Atomic Predicates.

Figs. 7d, 7e and 7f depict how the execution time of the Maximal Flow computation algorithm varies when the number of policies, NATs and firewalls respectively increases. In this case, there is no initial phase to compute the set of Atomic Predicates, but there is only the time that Algorithm 3 takes directly to generate the Maximal Flows.

Again, these figures allowed us to analyze the impact of each parameter of the network configuration for the performance of this algorithm. (i) *Number of policies* (Fig. 7d): The time increase is due to the fact that when there are more policies for which the corresponding Maximal Flows must be computed, Algorithm 3 has to run more times, one run for each policy. However, the time spent is significantly less compared to the one required by the Atomic Flow computation algorithm. (ii) *Number of NATs* (Fig. 7e): Even in the case of Maximal Flows, this is a determinant parameter. In fact, each NAT, and in general each transformer of the network, increases the number of generated flows. Each flow entering a transformer is split into multiple flows depending on the intersection the incoming predicate has with the transformation domain of the node. The more the transformers, the more the split flows and so the time to generate them. (iii) *Number of firewalls* (Fig. 7f). As for the number of NATs, this parameter determines an increase in time. This is explained by the fact that, when reaching a firewall, a predicate is intersected with the forwarding domain of the firewall (i.e., I^a and I^d). Therefore, for each existing intersection with one of these classes, a new Maximal Flow is generated.

Figs. 8a and 8b show the results of the tests executed by following the second strategy for both Atomic and Maximal Flow computation algorithms. This second method consists in the progressive enlargement of the network, and it was used to compare the two proposed approaches against the total time taken to generate the traffic flows and the total number of generated flows. The networks used as test cases are built starting from the same configurable parameters of the previous examples, with the difference that, in this case, all the parameters vary simultaneously. Specifically, 50 policies, 200 endpoints, 25 NATs and 25 FWs are added respectively to each test case. From the figures, the following main conclusions can be derived:

- Computing Atomic Flows requires more time than computing Maximal Flows, as shown in Fig. 8a. Most of the time is, in fact, spent in the initial phase to compute Atomic Predicates (Algorithm 1). On the contrary, Algo-



	pol	nodes	NAT	FW
A	100	400	50	50
B	150	600	75	75
C	200	800	100	100
D	250	1000	125	125
E	300	1200	150	150

(a) Flows computation time

	pol	nodes	NAT	FW
A	100	400	50	50
B	150	600	75	75
C	200	800	100	100
D	250	1000	125	125
E	300	1200	150	150

(b) Number of generated flows

Fig. 8: Tests on Atomic and Maximal Flows algorithms (second strategy)

rithm 3 for computing Maximal Flows is very fast, since it is a simple iterative function, mostly parallelizable.

- The solution with Atomic Flows generates a greater number of flows, as shown in Fig. 8b. This is motivated by the fact that the algorithm that computes Atomic Flows splits each flow into simple and minimal disjoint flows, while the one based on Maximal Flows aggregates them as much as possible.

From the results of these tests, it is not possible to see the real advantage of the Atomic Flows approach, which is to enable the representation of each predicate as a simple integer. In fact, this advantage is helpful in the subsequent network management tasks performed using the flows, which is object of validation in Subsection VII-B. As seen in Fig. 4, with the Atomic Flows approach, the NFs can map their operation over simple integers, while with the Maximal Flows approach more complex predicate representations are required.

B. Comparative validation for solving network security management problems

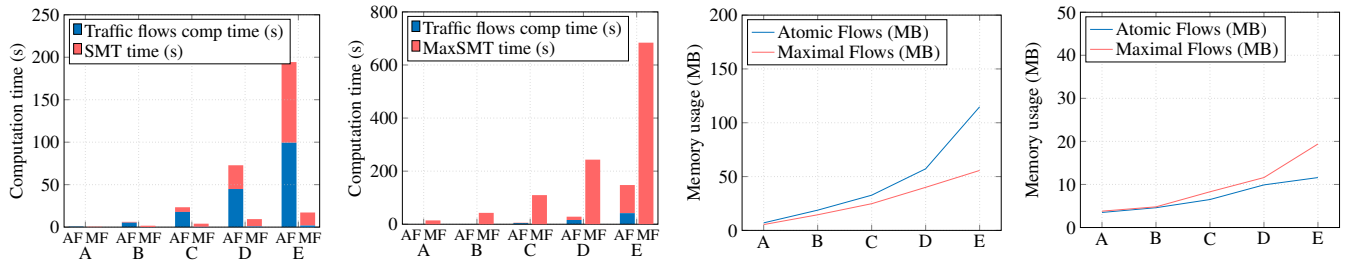
With the second class of tests, we evaluated the time and memory that two resolution algorithms for reachability verification and security policy refinement requires to solve their respective problems, when they work on the Atomic and Maximal Flow models.

Concerning reachability verification, we extended Verigraph2.0 [11], a framework designed to verify connectivity properties of a network, i.e, to verify whether a given set of reachability and isolation policies is enforced correctly in a network. Regarding policy refinement, instead, we extended Verefoo [13]: given a set of security policies, this tool is able to provide the automatic and optimal allocation and configuration of network security functions in order to satisfy the user-requested policies. Both tools formulate their respective network security problems with constraint programming, as a *Satisfiability Modulo Theories* (SMT) problem in the case of Verigraph2.0, and as a *Maximum* SMT (MaxSMT) in the case of Verefoo. In terms of computational complexity, SMT and MaxSMT are NP-complete [26]. Nevertheless, state-of-the-art solvers, such as the one used for the implementation of Verigraph2.0 and Verefoo (i.e., Z3 by Microsoft Research), can solve many instances of this problem in polynomial time with respect to the problem size [27].

To interface with the SMT/MaxSMT solver, a correct representation of the traffic flows (either Atomic or Maximal) has to be introduced. In particular, it is necessary to model the predicates included in the definition of traffic flow. As this kind of solvers typically understand only basic data types, the explicit representation of a predicate is critical. This is especially true for the Maximal Flows approach. In this case, the explicit representation of a predicate is the conjunction of four predicates defined on integer variables for the source IP address, of four predicates defined o integer variables for the destination IP address, two predicates defined on integer variables for the source port range, two predicates defined on integer variables for the destination port range, and a predicate defined on a string variable for the transport-level protocol. On the contrary, the approach using Atomic Flows only requires one integer constant for the predicate representing a packet class, i.e., the integer identifier representing the Atomic Predicate. The advantage of using Atomic Predicates, which was previously hidden, becomes apparent and results in a disparity of one integer against thirteen variables per predicate. However, it must also be said that the Maximal Flows approach generates a smaller number of flows (and so less predicates in input to each node), as we have seen in Fig. 8b. This reduction slightly mitigates the 1 vs 13 variables disparity. However, in general, imagining to split the total time to solve the network related problem into the time for computing traffic flows and the time to solve the SMT/MaxSMT problem, the Atomic Flows approach is expected to solve the second phase faster, because of the lower number of variables the solver has to manage. The Maximal Flows approach, instead, as seen in Fig. 8a, is faster in solving the traffic flows computation phase.

Again, the tests are carried out by simulating a progressive enlargement of the network. Each parameter is therefore incremented by a fixed value for each test case.

Fig. 9a shows the results to solve the reachability verification problem using Verigraph2.0 with the two proposed traffic flows approaches. For each test case the chart shows two bars: one on the left corresponding to the total time using Atomic Flows, one on the right corresponding to the total time using Maximal Flows. Each bar, in turn, is divided into two parts: the first one colored blue indicating the time required to compute the traffic flows and the second one colored orange indicating the time required to solve the SMT problem. In this case, the approach using Maximal Flows is advantageous over the approach that uses Atomic Flows. Looking at the time breakdown, the first phase to compute traffic flows turns out to be decisive for the Atomic Flows approach. Much of the time is, in fact, spent to compute the set of Atomic Predicates and, in addition, this approach generates a greater number of flows the solver must consider. So, even the SMT phase, in



	pol	nodes	NAT	FW
A	50	40	5	5
B	75	60	10	10
C	100	80	15	15
D	125	100	20	20
E	150	120	25	25

(a) Time - Reachability

	pol	nodes	NAT	FW
A	10	60	5	5
B	15	80	10	10
C	20	100	15	15
D	25	120	20	20
E	30	140	25	25

(b) Time - Refinement

	pol	nodes	NAT	FW
A	50	40	5	5
B	75	60	10	10
C	100	80	15	15
D	125	100	20	20
E	150	120	25	25

(c) Memory - Reachability

	pol	nodes	NAT	FW
A	10	60	5	5
B	15	80	10	10
C	20	100	15	15
D	25	120	20	20
E	30	140	25	25

(d) Memory - Refinement

Fig. 9: Comparative validation on synthetically generated networks

this case, is slower. Computing the Atomic Predicates does not pay off, because the traffic flows phase is longer and does not bring advantage to the subsequent SMT phase, which instead is solved very quickly even in the case of Maximal Flows, with the solver that has to manage complex predicates.

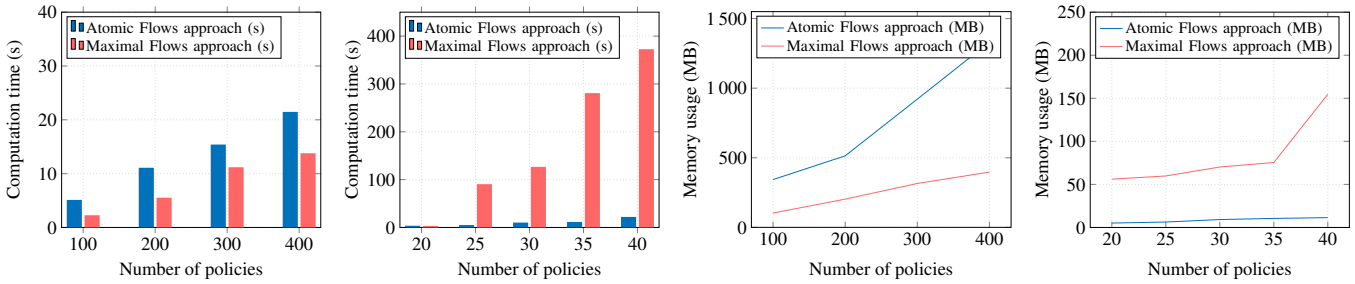
Fig. 9b, instead, shows the results to solve the Refinement problem in Verefoo. In this case, the approach using Atomic Flows performs much better than the one using Maximal Flows. Looking at the time breakdown, it can be seen that, as expected, the initial traffic flows computation phase using the Atomic Flows approach is slower in this case too, compared to the time required by the Maximal Flows approach. However, the subsequent MaxSMT phase is much faster. So, here, computing the Atomic Predicates and, then, using them for the MaxSMT problem helps to solve it much faster. Compared to the previous reachability verification process, in which the solver only had to check whether the connectivity policies were satisfied or not, here, in case of a Refinement process, the MaxSMT problem has much greater weight. In fact, it has to allocate and automatically configure the NSFs needed to satisfy the issued security policies. In this case, the MaxSMT phase has a greater weight than the traffic flows computation one (color orange in predominant in each bar over blue). Using Atomic Flows, the initial time spent to compute the set of Atomic Predicates brings enough advantage to make the resolution of the MaxSMT problem simpler and faster. So, spending more time for the initial phase pays off. The same MaxSMT problem is very slow using Maximal Flows. In the first case, the solver can work with integers and the whole network configuration problem becomes a simple process working on sets of integers. Conversely, with the Maximal Flows approach the solver has to work with a set of 13 variables for each predicate, and this increases the execution time.

Fig. 9c and Fig. 9d compare the memory usage of the newly developed Verigraph2.0 and Verefoo versions based on Atomic Flows and the original versions based on Maximal Flows. The results graphically represented in these figures show that solving the reachability verification problem with Verigraph2.0 requires more memory if Atomic Flows are used, while solving the security policy refinement problem with Verefoo requires more memory if Maximal Flows are used. The former result is explained by the fact that the number of Atomic Flows that is produced is much higher than the number of Maximal Flows. Therefore, even if the Maximal Flows approach requires a set of 13 variables for each predicate representing a packet class, the memory required to store this higher number of variables does not overcome the memory required to store information about all Atomic Flows, also because in the SMT problem embedded in Verigraph2.0 there are no optimization constraints defined on each one of those variables. Instead, the latter result is explained by the fact that in the MaxSMT problem formulated in Verefoo there are optimization constraints defined on all variables of all predicates representing packet classes. Consequently, creating and storing all these constraints determines a memory usage that is higher by the one requested in case of the Atomic Flow strategy, where more flows are created, but there are less optimization constraints, as each packet class is simply identified by a single integer variable. Moreover, it is worth noticing that these results are in line with the ones about time scalability. In fact, Maximal Flows were proved to be more efficient for solving the reachability verification problem also in terms of computation time, and the same applies to Atomic Flows for solving the security policy refinement problem.

All the considerations made so far were confirmed also by the tests we performed on network configurations inspired by GÉANT[3], Internet2[4] and APAN[5], three existing real production networks that are located respectively in Europe, America and Asia. The tests on these network topologies were carried out by varying the number of security policies, and assessing

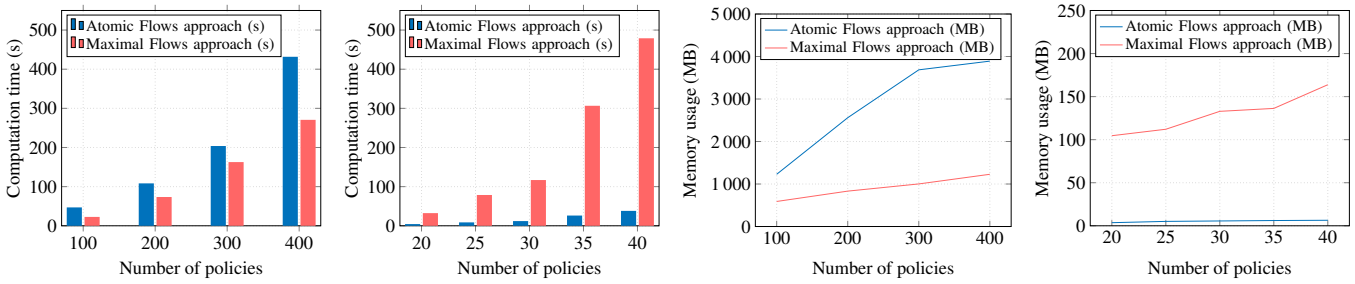
[3]https://network.geant.org/
[4]https://internet2.edu/network/
[5]https://apan.net/

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2024.3407159



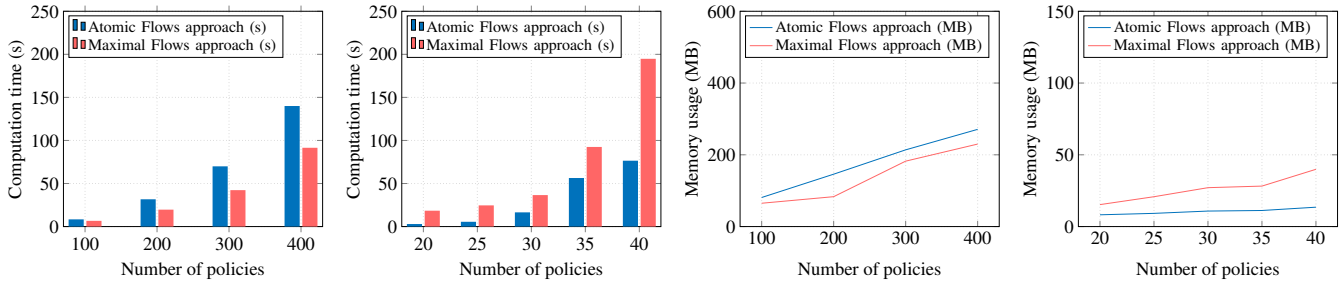
(a) Time - Reachability (b) Time - Refinement (c) Memory - Reachability (d) Memory - Refinement

Fig. 10: Comparative validation on a network inspired by the GÉANT topology



(a) Time - Reachability (b) Time - Refinement (c) Memory - Reachability (d) Memory - Refinement

Fig. 11: Comparative validation on a network inspired by the Internet2 topology



(a) Time - Reachability (b) Time - Refinement (c) Memory - Reachability (d) Memory - Refinement

Fig. 12: Comparative validation on a network inspired by the APAN topology

computation time and memory usage of both variants of the two frameworks: the original Maximal Flow variants, and the newly developed Atomic Flow ones. Their results have been graphically reported in Fig. 10, Fig. 11, and Fig. 12. Specifically:

- Fig. 10a, Fig. 11a and Fig. 12a report the total computation time required by the two Verigraph2.0 variants to solve the reachability verification problem;
- Fig. 10b, Fig. 11b and Fig. 12b report the total computation time required by the two Verefoo variants to solve the refinement problem;
- Fig. 10c, Fig. 11c and Fig. 12c report the memory usage of the two Verigraph2.0 variants to solve the reachability verification problem;
- Fig. 10d, Fig. 11d and Fig. 12d report the memory usage of the two Verefoo variants to solve the refinement problem.

All these experimental results demonstrate again that the approach using Maximal Flows requires less time and memory to solve the reachability verification problem, while the approach using Atomic Flows is more convenient in solving the refinement problem.

VIII. Conclusions

This paper introduced a two-fold traffic flow model, which can be used as a starting point to pair formal methods with automation for network security management. On this general model, two different algorithms have been built to group packet classes into traffic flows entities, named Maximal and Atomic Flows. These algorithms differ for the trade-off they can achieve between two features of their grouping strategies, i.e., the number of traffic flows that are computed, and the

granularity level of each traffic flow. Slightly modified state-of-the-art automatic algorithms can employ these flow entities to solve different network security management problems such as security policy verification and refinement.

An extensive validation of the algorithm execution and their application to management problems was carried out to assess their feasibility, and to identify the cases where each algorithm performs better. From the validation results, we noticed that aggregating packet classes into Maximal Flows is more convenient to solve less burdensome problems, such as the verification of reachability and isolation policies, because the time and memory required to compute the flows are lower. Instead, using Atomic Flows is advisable when addressing more complex problems such as policy refinement. Even if more time is needed to compute the Atomic Predicates at the basis of these flows, each one can be assigned an integer identifier, and these numbers can be used to formulate the problem in a simpler way, using less variables.

Future work envisions to assess the feasibility of this traffic flow model in other network security management problems , such as formal verification related to information disclosure, latency constraints, and reliability. Different trade-offs between number of computed flows and their granularity level will be also investigated, to understand how they can impact the resolution of these problems.

ACKNOWLEDGMENT

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

REFERENCES

[1] S. Scott-Hayward, S. Natarajan, and S. Sezer, "A survey of security in software defined networks," *IEEE Commun. Surv. Tutorials*, vol. 18, no. 1, pp. 623–654, 2016.

[2] R. Boutaba and I. Aib, "Policy-based management: A historical perspective," *J. Netw. Syst. Manag.*, vol. 15, no. 4, pp. 447–480, 2007.

[3] A. Leivadeas and M. Falkner, "A survey on intent based networking," *IEEE Commun. Surv. Tutor.*, 2022.

[4] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 57:1–57:37, 2024.

[5] J. Qadir and O. Hasan, "Applying formal methods to networking: Theory, techniques, and applications," *IEEE Commun. Surv. Tutorials*, vol. 17, no. 1, pp. 256–291, 2015.

[6] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proc. of the {USENIX} Symp. on Net. Syst. Design and Impl.*, 2013.

[7] J. Govaerts, A. K. Bandara, and K. Curran, "A formal logic approach to firewall packet filtering analysis and generation," *Artif. Intell. Rev.*, vol. 29, no. 3-4, 2008.

[8] P. Bera, S. K. Ghosh, and P. Dasgupta, "Policy based security analysis in enterprise networks: A formal approach," *IEEE Trans. Netw. Service Manag.*, vol. 7, no. 4, 2010.

[9] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Trans. on Net.*, vol. 24, no. 2, 2016.

[10] ——, "Scalable verification of networks with packet transformers using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 2900–2915, 2017.

[11] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, "Improving the formal verification of reachability policies in virtualized networks," *IEEE Trans. on Net. and Serv. Manag.*, vol. 18, no. 1, 2020.

[12] M. A. Rahman and E. Al-Shaer, "Automated synthesis of distributed network access controls: A formal framework with refinement," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 2, 2017.

[13] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1559–1576, 2023.

[14] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang, "A survey on network verification and testing with formal methods: Approaches and challenges," *IEEE Commun. Surv. Tutorials*, vol. 21, no. 1, pp. 940–969, 2019.

[15] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Towards a fully automated and optimized network security functions orchestration," in *2019 4th International Conference on Computing, Communications and Security (ICCCS), Rome, Italy, October 10-12, 2019*. IEEE, 2019, pp. 1–7.

[16] S. Bussa, R. Sisto, and F. Valenza, "Security automation using traffic flow modeling," in *8th IEEE International Conference on Network Softwarization, NetSoft 2022, Milan, Italy, June 27 - July 1, 2022*. IEEE, 2022, pp. 486–491.

[17] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of ip networks," in *Proc. of the IEEE Conf. of the IEEE Comp. and Comm. Societies.*, 2005.

[18] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. of the {USENIX} Symp. on Net. Syst. Design and Impl.*, 2012.

[19] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. of the {USENIX} Symp. on Net. Syst. Design and Impl.*, 2013.

[20] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. of the {USENIX} Symp. on Net. Syst. Design and Impl.*, 2015.

[21] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proc. of the ACM SIGCOMM Conference*, 2016.

[22] N. Stouls and M. Potet, "Security policy enforcement through refinement process," in *Proc. of the 7th Intern. Conf. of B Users, Besançon*, 2007.

[23] D. Ranathunga, M. Roughan, P. Kernick, and N. Falkner, "The mathematical foundations for mapping policies to network devices," in *Proc. of the 13th Intern. Joint Conf. on e-Business and Telecommunications*, 2016.

[24] N. Schnepf, R. Badonnel, A. Lahmadi, and S. Merz, "Rule-based synthesis of chains of security functions for software-defined networks," *ECEASST*, vol. 76, 2018.

[25] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks," in *Proc. of the IEEE/IFIP Network Operations and Management Symp.*, 2020.

[26] M. E. Halaby, "On the computational complexity of maxsat," *Electron. Colloquium Comput. Complex.*, 2016.

[27] R. Robere, A. Kolokolova, and V. Ganesh, "The proof complexity of SMT solvers," in *Computer Aided Verification*. Springer International Publishing, 2018.

Daniele Bringhenti received the M.Sc. degree (summa cum laude) and the Ph.D. degree (summa cum laude) in computer engineering from the Politecnico di Torino, Torino, Italy, in 2019 and 2022 respectively, where he is currently an Assistant Professor with time contract. His research interests include novel networking technologies, automatic orchestration and configuration of security functions in virtualized networks, formal verification of network security policies.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2024.3407159

Simone Bussa received the M.Sc. degree (summa cum laude) in computer engineering from the Politecnico di Torino, Turin, Italy, in 2021, where he is currently working toward the Ph.D. degree in control and computer engineering. His research interests include distributed systems security and formal verification applied in the field of cyber-physical systems.

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.

Fulvio Valenza received the M.Sc. degree (summa cum laude) and the Ph.D. degree (summa cum laude) in computer engineering from the Politecnico di Torino, Torino, Italy, in 2013 and 2017, respectively, where he is currently a Tenure-Track Assistant Professor. His research activity focuses on network security policies, orchestration and management of network security functions in SDN/NFV-based networks, and threat modeling.