Security Automation using Traffic Flow Modeling

Security Automation using Traffic Flow Modeling

Simone Bussa, Riccardo Sisto, Fulvio Valenza
Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy, Emails: {first.last}@polito.it

***Abstract*—The growing trend towards network "softwarization" allows the creation and deployment of even complex network environments in a few minutes or seconds, rather than days or weeks as required by traditional methods. This revolutionary approach made it necessary to seek automatic processes to solve network security problems. One of the main issues in the automation of network security concerns the proper and efficient modeling of network traffic. In this paper, we describe two optimized Traffic Flows representation models, called Atomic Flows and Maximal Flows. In addition to the description, we have validated and evaluated the proposed models to solve two key network security problems - security verification and automatic configuration - showing the advantages and limitations of each solution.**

***Index Terms*—Traffic modeling, Security Automation, Network Security Functions**

I. Introduction

Nowadays, computer networks continue to grow in size and importance. Emerging technologies such as Software-Defined Networking and Network Functions Virtualization have introduced dynamism and flexibility in networking, making the configuration and verification of network security critical [1], [2]. The typical manual management approach, frequently based on trial-and-error, can no longer be used in these scenarios. These techniques, besides being cumbersome and time-consuming, lack a comprehensive view of the network behavior and, as such, are error-prone and can make the network security significantly hard to maintain. In view of these factors, automatic processes and tools that configure and verify the correctness of network security in a formal and deterministic way are becoming crucial, as they enable the automatic prevention of the violation of security properties and service downtimes [3], [4].

A key problem in the automation of network security management based on formal approaches is the proper and efficient modeling of network traffic, which is the basis for modeling network security properties. Formally modeling how the packets originating in the source of a communication can be actually forwarded and modified in a network is a complex task, which involves modeling: (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.

Unfortunately, in the literature, only few studies that use formal network traffic models for automatic network security management are available [3], [4], [5], [6], [7], [8], and they are mostly limited to simple networks, which include packet filters and simple transformers, and only to reliability verification. We are therefore still far from a context of real networks made up of a large number of network security functions (NSFs) and with the goal to automatically configure and/or verify them. The state of the art mostly fails against those challenges mainly for reasons of efficiency and scalability.

In this paper, we characterize and compare two novel and optimized approaches, called respectively Atomic Flows and Maximal Flows, to represent, identify, and aggregate network traffic, as required in the processes for formal automatic security management. Specifically, we analyze the advantages and limitations of the two models, to solve two main problems: automatic security configuration and verification. Having defined the two approaches, thus, we implemented each of them in two existing frameworks, Verigraph [3] and Verefoo [9], which aim to solve respectively a classical Reachability and Refinement problem.

The remainder of this paper is structured as follows. In Section II, we introduce some key concepts about network traffic modeling. In Section III and IV, we present the Atomic Flows and Maximal Flows model. In Section V, we make a detailed comparison between the two approaches. Finally, in Section VI we give a summary of related work and Section VII we present the conclusions and future work.

II. Approach

As mentioned in the introduction, in this paper we describe two different and optimized approaches, called Atomic Flows and Maximal Flows, to identify and aggregate the network traffic. Before getting to the core of the discussion, we must first introduce some general concepts about how we assume the network and its security properties are modeled.

A. Network Model

We consider the modeling approaches that define the network as a graph whose nodes represent the network functions and endpoints (i.e., web clients, web servers, routers, firewalls, VPN gateways, NATs, etc). Each node may have a set of input and a set of output ports, each port could be controlled by an ACL, which describes whether a packet with a certain header can pass through that port or not. We denote I_a the set of packets allowed to pass, and I_d the set of those denied. We call these sets the domains of the port. If an output port is not controlled by an ACL, this means that all traffic can pass through that port, i.e., I_a is the set of all packets while I_d is empty. The packets that enter a function and that cannot be transferred to any output port are discarded by the function.

A packet that enters through an input port could be transformed before being transmitted through an output port. The most common transformations are header rewriting, encapsulation and de-encapsulation. The transformation behavior is modeled by a function T, which is characterized by a set of input domains each one corresponding to a set of output domains. For a forwarder that simply forwards packets without modifying them, for example, function T has a single input domain D, that matches all the packets, T is the Identity function, and the output domain is D as well. For a NAT, instead, there are three main domains, D_1, D_2, D_3, undergoing three distinct transformations: a packet that matches with D_1 is affected by the Shadowing operation, one that matches with D_2 by the Reconversion operation, whereas for one that matches with D_3 no transformations are applied, and the packet is simply forwarded (or discarded, depending on the NAT configuration policy). The choices of forwarding and transforming a packet are made based on some packet contents, usually some header fields. Hence, we model packets only considering such packet fields. Predicates over these fields are used to model classes of packets.

Definition II.1. Predicate: A class of packets, also called traffic, is modeled as a predicate defined over variables that represent some packet fields. In this way, packets that do not differ in these fields are represented by the same class.

There are several ways to represent a predicate (i.e., BDD, Tuple Representation [10], Wildcards Expressions [11], FDD [12], etc.). In this work, we reuse the approach introduced in [3]: a predicate is the conjunction of sub-predicates, one for each packet field that is considered, and this conjunction is denoted by the tuple of its sub-predicates. Here, for simplicity, we consider IP packets, modeled by predicates over the IP quintuple (IP source, IP destination, port source, port destination, protocol type). We also consider that each sub-predicate can represent ether a single value, or a range of values, or the full range, denoted by the wildcard "*". For example, a valid tuple is (10.0.0.1, 30.0.5.*,80, *, tcp).

B. Traffic flows

Having defined the predicates that represent packet classes, let us now define the concept of Traffic Flow. In analogy with [3], a traffic flow represents a flow of packets that can cross a network. The traffic flow model describes how a certain class of packets is forwarded and transformed from its source to its destination.

Definition II.2. Traffic flows: A Traffic Flow $f \in F$ 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 NSFs. The main advantage of this approach, compared to the alternative modeling approaches, is that the NSFs 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.

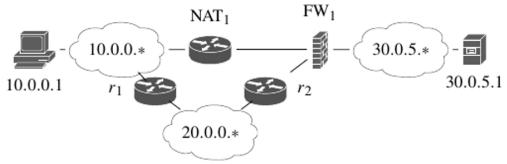
The definition of traffic flows given above allows some freedom concerning the granularity of flows, i.e., it is possible to consider fewer flows characterized by larger packet classes or more flows characterized by simpler packet classes. We need modeling solutions that enable efficient and scalable refinement and verification algorithms. However, which approach is better to achieve this goal is difficult to tell, as both the number of flows and their complexity may have an impact on the complexity of such algorithms. In this work, therefore, we propose and study two different approaches for the identification and computation of flows. The first one, called Atomic Flows, simplifies the representation of packet classes as much as possible, leading to many simple flows, while the second one, called Maximal Flows, reduces the number of flows, aggregating as much as possible several flows together, leading to few but complex flows.

C. Network Security Policy

When we want to model network traffic to verify or refine some security properties, we are interested in a subset of all traffic flows that are possible in the network. The interesting flows can be selected by considering only certain sources and destinations for the flows, according to the Security Policies. For example, a security policy about isolation could be expressed by a set of rules, each one taking the form $p = (C, a)$, where a is the action to perform on network packets that match the condition C. From the condition C of a rule, we can extract a quintuple predicate t expressing the source-destination pair for our flow: t = (IPSrc, IPDst, pSrc, pDst, tProto), where: 1) the source and destination nodes of the flow have IP addresses matching respectively t.IPSrc and t.IPDst, 2) the source traffic satisfies t.IPSrc and t.pSrc ((t.IPSrc, *, t.pSrc, *, *)), 3) the destination traffic satisfies t.IPDst, t.pDst and t.tProto ((*, t.IPDst, *, t.pDst, t.tProto)). Example: (10.0.0.1, 30.0.5.1, 200, 80, TCP) is a condition on TCP traffic that exits from port 200 of node 10.0.0.1 and is directed to port 80 of node 30.0.5.1.

D. Running example

To better understand the proposed approaches, we will use the simple scenario described in Fig. 1 as a running example. Specifically there are two sub-nets of clients (10.0.0.0/24 and 20.0.0.0/24) and a server (30.0.5.2 in sub-net 30.0.5.0/24). There is a NAT just outside the 10.0.0.0/24 sub-net, performing the Shadowing operation only on IP address 10.0.0.1, and a firewall with its own Access List, containing only one rule. In this scenario we want to compute the traffic flows linking the subnet of clients 10.0.0.* with the server 30.0.5.1, without any limitation on the port numbers and protocol type. The



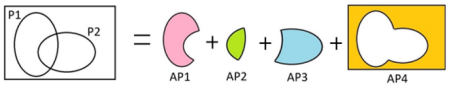
FW1 Access List						
#	Action	IPSrc	IPDst	pSrc	pDst	tProto
1	Allow	10.0.0.1	30.0.5.1	*	*	ANY
D	Deny	*	*	*	*	ANY
I_a = {10.0.0.1, 30.0.5.1, *, *, ANY}						
I_d = $\neg I_a$						

TABLE A. *Forwarding domain for FW1*

NAT1 30.0.0.1	
Sources	10.0.0.1
D_1 (Shadowing)	{10.0.0.1, ¬10.0.0.1∧ ¬30.0.0.1, *, *, ANY}
D_2 (Reconversion)	{¬10.0.0.1∧ ¬30.0.0.1, 30.0.0.1, *, *, ANY}
D_3 (Simple Forwarding)	$\neg D_1 \wedge \neg D_2$

TABLE B. *Input transformation domains for NAT1*

Fig. 1: Example scenario



P1 = {10.0.0.*, 30.0.5.1, *, *, ANY}	P2 = {10.0.0.1, 30.0.5.*, *, *, ANY}
AP1 = {10.0.0.* ∧ ¬10.0.0.1, 30.0.5.1, *, *, ANY}	AP2 = {10.0.0.1, 30.0.5.1, *, *, ANY}
AP3 = {10.0.0.1, 30.0.5.* ∧ ¬30.0.5.1, *, *, ANY}	AP4 = {¬10.0.0.*, *, *, *, ANY} U {*, ¬30.0.5.*, *, *, ANY}

Fig. 2: From predicates to atomic predicates

quintuple expressing this condition is (10.0.0.*, 30.0.5.1, *, *, ANY). The corresponding flows can take two paths, either passing through the NAT or through the 20.0.0.0/24 subnet.

III. Atomic Flows

The basic idea behind the Atomic Flow approach is based on the concept of Atomic Predicate, first presented in [8]. Given a set of predicates of the network, it is possible to compute the set of totally disjunct and minimal predicates (Atomic) such that each predicate of the first set can be expressed as a disjunction of a subset of those in the second set. In other words, it is possible to split each complex predicate (representing for example a NAT input class, source traffic, etc) into a set of simpler and minimal Atomic Predicates. An example is shown in Fig. 2. Starting from P1 (predicate representing packets traveling from the subnet 10.0.0.0/24 to 30.0.5.1) and P2 (predicate representing packets from 10.0.0.1 to the subnet 30.0.5.0/24), it is possible to compute the set of corresponding Atomic Predicates.

From the set of Atomic Predicates we can compute the set of Atomic Flows.

Definition III.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.

Since the atomic predicates are disjoint and unique within the network, the advantage is that it is possible to assign them simple identifiers (e.g., integers) and then use only these identifiers in all computations, instead of using the more complex explicit representations (BDD, Tuple Representation, etc.). The goal is to split each traffic flow into sub-flows that are as simple as possible and totally disjoint from one another, so that it is possible to replace each predicate, defined within the flow, with its identifier.

In developing our approach, we took a cue from algorithms described in paper [8], that defines how to compute the set of atomic predicates starting from a general network predicates.

A. Atomic Flows Example

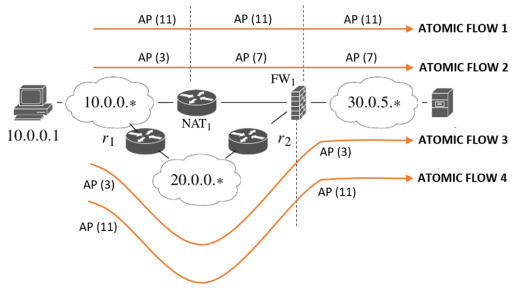
Let us now see, through the running example, how the approach with Atomic Flows works (Fig. 3).

First, we need to compute the set of Atomic Predicates, limiting them to only those related to the security policies that we want to investigate. As we said, we want to compute all the traffic flows linking the subnet 10.0.0.* with the server 30.0.5.1. Therefore, we must generate the atomic predicates corresponding to source traffic (10.0.0. *, *, *, *, ANY) and destination traffic (*, 30.0.5.1, *, *, ANY), plus all the predicates included in the domains of the network functions found along the paths. The network functions crossed by at least one path are NAT and FW, and the corresponding input domains can be seen in Table A and Table B of Fig. 1. In addition, the set of atomic predicates must also include the predicates resulting from possible transformations. The final resulting set of atomic predicates, each one associated with its integer identifier, can be seen in Table A of Fig. 3.

Having defined the set of atomic predicates, we can model the forwarding and transformation behavior of the NSFs as functions working on integers. This is the main advantage of this approach. The resulting functions can be seen in tables B and C of Fig. 3. We can see, for example, how the atomic predicate AP(3), which arrives in input to the NAT, matches its input domain D1 and is transformed into AP(7), or how the same AP(3) is the only predicate allowed to pass through the FW. At this point, we can compute the Atomic Flows. We start from the set of atomic predicates representing the traffic generated by the source 10.0.0.*, that is AP(1) to AP(4) and AP(9) to AP(12). Each single atomic predicate is propagated along each path linking source to destination, taking into account that, crossing a node, it can be transformed into one or more different disjoint atomic predicates or it can be dropped. Moreover, some generated Atomic Flows can be discarded because they are not part of the solution (i.e., they do not arrive at destination with the correct IPDst, pDst and tProto) or because they have reached the destination without reaching the destination of the path (i.e., their destination is an intermediate node). As the result, we have four Atomic Flows. As we can see, they are all disjoint and they are simple alternation of nodes and integer identifiers of the atomic predicates.

IV. Maximal Flows

The second approach we propose makes use of Maximal Flows, and originates in a concept exactly opposite to that of Atomic Flows. Whereas with Atomic Flows we tried to split as much as possible each traffic flow, so that it contains only minimal and disjoint traffics that can be subsequently identified with an integer identifier, reaching the highest level of granularity but also a higher number of flows, with this



1) {10.0.0.1, 10.0.0.1, *, *, ANY}	9) {10.0.0.*∧ ¬10.0.0.1, 10.0.0.1, *, *, ANY}
2) {10.0.0.1, 30.0.0.1, *, *, ANY}	10) {10.0.0.*∧ ¬10.0.0.1, 30.0.0.1, *, *, ANY}
3) {10.0.0.1, 30.0.5.1, *, *, ANY}	11) {10.0.0.*∧ ¬10.0.0.1, 30.0.5.1, *, *, ANY}
4) {10.0.0.1, ¬10.0.0.1∧ ¬30.0.0.1∧ ¬30.0.5.1, *, *, ANY}	12) {10.0.0.*∧ ¬10.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1∧ ¬30.0.5.1, *, *, ANY}
5) {30.0.0.1, 10.0.0.1, *, *, ANY}	13) {¬10.0.0.*∧ ¬30.0.0.1, 10.0.0.1, *, *, ANY}
6) {30.0.0.1, 30.0.0.1, *, *, ANY}	14) {¬10.0.0.*∧ ¬30.0.0.1, 30.0.0.1, *, *, ANY}
7) {30.0.0.1, 30.0.5.1, *, *, ANY}	15) {¬10.0.0.*∧ ¬30.0.0.1, 30.0.5.1, *, *, ANY}
8) {30.0.0.1, ¬10.0.0.1∧ ¬30.0.0.1∧ ¬30.0.5.1, *, *, ANY}	16) {¬10.0.0.*∧ ¬30.0.0.1, ¬10.0.0.1 ∧ ¬30.0.0.1∧ ¬30.0.5.1, *, *, ANY}

TABLE A. *Final Atomic Predicates with their integer identifier*

D1, Shadowing	(3) becomes (7), (4) becomes (8)
D2, Reconversion	(10) becomes (9), (14) becomes (13)
D3, Simple Forwarding	(1), (5), (11), (12), (15), (16) are simply forwarded (2), (6) reach their destination in the NAT

TABLE B. *Transformations for NAT1*

Ia	(3) allowed to pass
Id	(1), (2), (4), (5), (6), (7), (8), (9), (10), (11), (12), (13), (14), (15), (16) dropped

TABLE C. *Forwarding domain of FW1*

Fig. 3: Atomic Flows

approach we try to do the opposite by aggregating flows together. That is, we try to reduce the number of generated flows, considering only a subset of them, which is smaller but equally representative: the set of Maximal Flows. All flows represented by the same maximal flow (we could indicate them as subflows of the maximal flow) behave in the same way when crossing the network, so that it is sufficient to consider the maximal flow and not each single flow that it represents.

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

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

The set F_r^M is defined as a subset of F_r that contains only the flows that are not subflows of any other flow in F_r. In other words, we aggregate in the same Maximal Flow all the flows behaving in the same way, and then we consider for our analysis only F_r^M which has a smaller size than F_r.

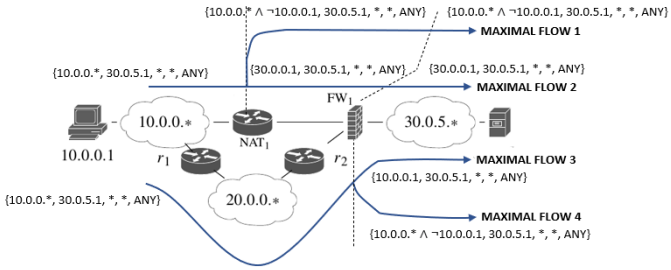
Since Predicates within a Maximal Flow are the result of aggregating together multiple traffics, they are a disjunction of several quintuples and, for this reason, they cannot be Atomic. In this case they cannot be replaced by integer identifiers but they have to be represented by other more complex data structures (BDD, Wildcards Expressions, etc.).

Further details on Maximal Flows and the algorithms to compute them can be found in our previous work [9].

A. Maximal Flows Example

Analyzing the results in Fig. 4, we see that each flow begins with a traffic that is as general and inclusive as possible (the largest traffic that satisfies the security policy we are considering). The basic idea, in fact, is to start with a large maximal flow (which includes as many subflows as possible) and then divide this flow into smaller flows only when necessary, for example when it encounters a function for which at least two sub-flows (of the large flow we are considering) have different behaviors.

This happens, for example, in the NAT and in the FW. In the NAT, the incoming flow is divided into two new flows: the first one starting from predicate (10.0.0.1, 30.0.5.1, *, *, ANY) which undergoes the shadowing operation and is transformed into (30.0.0.1, 30.0.5.1, *, *, ANY), and the second one which is forwarded unchanged. The main flow in this way is split into two different flows. The same happens considering the FW and its forwarding domains. Also in this case the incoming flow is split into two new flows: the first one matching with Ia, and the second one matchig with Id.



{10.0.0.*, 30.0.5.1,*, *, ANY} *intersects NAT1 transformation domain and it is split into* {10.0.0.* ∧ ¬10.0.0.1, 30.0.5.1, *, *, ANY} *and* {10.0.0.1, 30.0.5.1, *, *, ANY}. *This last predicate is then Shadowed into* {30.0.0.1, 30.0.5.1, *, *, ANY}
{10.0.0.*, 30.0.5.1, *, *, ANY} *intersects the FW1 rule #1 and it is split into* {10.0.0.1, 30.0.5.1, *, *, ANY} *and* {10.0.0.* ∧ ¬10.0.0.1, 30.0.5.1, *, *, ANY}

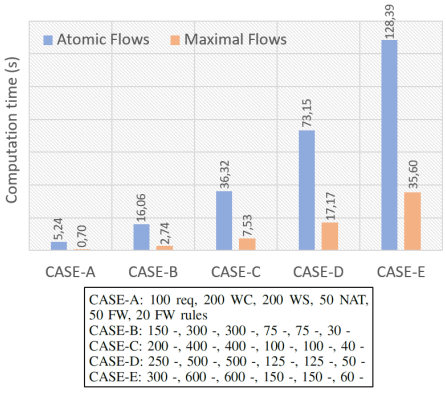
Fig. 4: Maximal Flows

V. Discussion and Comparison

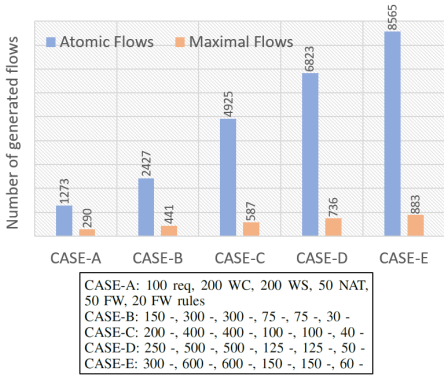
In this section, we first evaluate the single performance to compute the Atomic Flows and Maximal Flows, then we analyze the advantages and limitations of each approach applied in existing automatic security management tasks, such as security verification and configuration.

In this evaluation, the networks used as test cases differ in a set of configurable parameters: number of security policies intended as pairs of sources and destinations (req), number of web clients present in the network (WC), number of web servers (WS), number of NATs (NAT), number of firewalls (FW) and number of rules configured within each firewall. Moreover, each approach proposed in this paper has been implemented in Java.

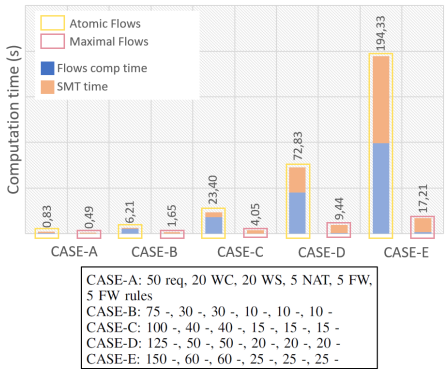
Results of the first analysis are shown in Fig. 5.a (time taken to compute the set of flows) and Fig. 5.b (number of generated flows). Looking at the two charts, we can say that: (i) The solution for the computation of the Atomic Flows is slower than the one of the Maximal Flows. Most of the time is spent on the initial computation of the set of Atomic Predicates. The algorithm for computing Maximal Flows, on the contrary, is a simple recursive function mostly parallelizable, so it is very fast; (ii) The solution with Atomic Flows generates a greater number of flows. This can be easily understood since with Atomic Flows we split each flow into minimal flows that are



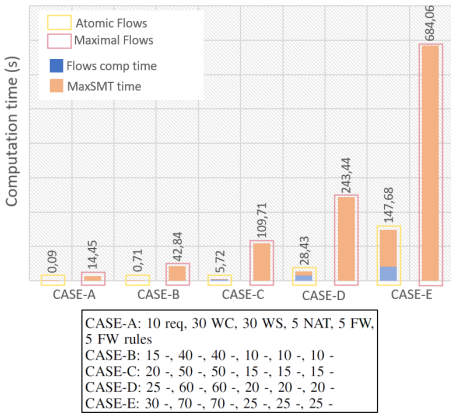
(5.a) Traffic Flows computation time



(5.b) Number of generated flows



(5.c) Time Reachability Verigraph2.0



(5.d) Time Refinement Verefoo

simpler and disjoint, while with Maximal Flows we try as much as possible to aggregate them; (iii) Finally, the advantage of Atomic Flows is that they allow to represent each predicate as a simple integer, while with the Maximal Flows approach this is not possible. As we have seen in Fig. 3, with the Atomic Flows approach, the NSFs can work with the integer identifier of each predicate, while using Maximal Flows they have to work with the complex explicit representation of it.

As already introduced, the next step was to use the two approaches to solve two network security-related problems: policy refinement and verification, where an optimized definition of traffic flow plays a crucial role. Specifically, Policy refinement is the process that translates high-level policies into low-level system configurations [9]. This is a critical task for the system security that, if not carefully performed, may lead to either incorrect or sub-optimal implementations. Computer-aided automatic tools are therefore needed to assist administrators in the translation of network security policies, because when this translation is performed by hand the risk of errors increases. On the other hand, Policy verification deals with checking whether a set of policies is correctly enforced in a system [3]. Typically, Policy verification is used to validate an hand-made refinement, to detect possible anomalies derived from the policy translation. Definitely, Policy Refinement and/or Verification should be done every time there is a change in the network policies and building automatic tools that assist the administrator can really help to speed up the process and increase reliability at the same time.

Concerning the verification process, to evaluate our novel approaches, we extended Verigraph2.0 [3], a framework that aims to solve Reachability problems, i.e., verifying if different nodes in the network can communicate or not. While for the refinement, we extended the framework Verefoo [9]: given a number of Security Policies (Reachability or Isolation between two endpoints), the framework is able to provide the automatic allocation and configuration of packet filters so that all the policies are satisfied, if possible. Both the proposed frameworks use an SMT solver (MaxSMT in the case of Refinement) and processes consist of two phases: a first phase for the computation of the Traffic Flows and a second phase in which the solver tries to find a solution that satisfies all the requirements, reasoning with the previous generated flows.

Using the solver requires modeling the predicates defined within each flow. This is particularly critical for the approach using Maximal Flows. As the solver can only work with basic data types, in this case, the explicit representation of a predicate becomes: 4 integers for source IP, 4 for destination IP, 2 integers for source port range, 2 for destination port and a string for protocol type. On the contrary, with the approach using Atomic Flows, the SMT (MaxSMT) solver works with simple integers, each one identifying a predicate. This results in a disparity 1 integer VS 13 variables per predicate. A possible advantage of the Maximal Flows approach, that slightly mitigates this disparity, is due to the fact that it generates a smaller number of flows, as we have seen in Fig. 5.b. Therefore, even if the solver requires 13 variables to represent each predicate, predicates are fewer in number. However, in general, we expect the SMT (MaxSMT) phase to be solved more quickly with the Atomic Flows approach. While for the first phase, as we have seen in Fig. 5.a, the approach with Maximal Flows is much more performing.

As we can see from Fig. 5.c, to solve the Reachability problem in Verigraph2.0, the approach using Maximal Flows (right bar for each test case) is advantageous over the Atomic Flows one (left bar). The initial time spent to compute the Atomic Predicates turns out to be decisive. They also introduce more

flows and requirements the solver must consider. If, instead, we analyze the example of the Refinement problem in Verefoo (Fig. 5.d), we see that the approach using Atomic Flows is much more performing than the one using Maximal Flows. This can be explained by the fact that, in the Reachability problem, the SMT phase (colored orange) has a relatively equal weight with respect to the first flows computation phase (colored blue). The SMT solver has only to check if the configurations of the network meet the security policies. Hence, the SMT phase is solved quickly, and the initial phase of flows computation is decisive. And this is disadvantageous for the Atomic Flows approach. For the Refinement problem, on the other hand, analyzing Fig. 5.d, we can see that the MaxSMT problem has a much greater weight in both approaches with respect to the first phase of flow computation (color orange in predominant in each bar over blue). The solver, in fact, has to allocate and automatically configure the packet filters needed to satisfy the issued security policies. In this case, using Atomic Flows, the initial time spent to compute the set of Atomic Predicates brings enough advantage to make leaner and faster the resolution of the MaxSMT problem, which is very slow using Maximal Flows.

VI. Related Work

A first study on the possibility of using automatic tools to verify network properties was presented here [13]. This work proposed a new approach to check in a static way the reachability between end-points of a network, by looking at the configuration state of routers, firewalls and stateless transformers. This was followed by many other researches, to find algorithms that verify network properties (not just reachability): [5], [4], [6], [7], [8], [3]. For all these works, the main issue is how to efficiently model the network and the network functions.

HSA [5] (and its successor [11]) models packets as points in the *Header Space* and network nodes as functions transforming a packet from one point to another point (or points) in that space. The behavior of a packet along a path (that is what we consider a flow) is obtained by composing the functions of the nodes it crosses. Veriflow [4] deals with disjoint *Equivalent Classes* (EC) of packets, that are sets of packets experiencing the same forwarding actions throughout the network. For each EC, the tool builds a Forwarding graph representing the paths this EC can take according to the forwarding behavior of the network. NoD [6] models headers fields as separate variables (not just a flat sequence of 0s and 1s as in [5]) and then expresses network functions as Datalog rules. Symnet [7] uses symbolic execution to perform static analysis. The novelty of this work is SEFL, a programming language used to express the data plane of the network in a symbolic-execution friendly manner. SEFL is used to model packets and network functions later used by Symnet to perform the symbolic execution. Another work that particularly inspired us is APVerifier [8], in particular for the concept of Atomic Predicates explained in section III. This work shows the benefits of working with minimal and disjoint predicates that can be replaced by integer identifiers. Operations on complex predicates, each one represented by a disjunction of atomic predicates, become operations on a set of integers.

In general, all the works mentioned above define network and traffic models to solve specific tasks (i.e., reachability, loop-freenes, cross-consistency etc.) and cannot be used to solve other network-related problems. The novelty of our proposed approach is that it is general enough to be applied to different contexts and uses, as we have seen in section V, to solve both verification and configuration problems.

VII. Conclusions

We resumed the definition of Traffic Flows that allow to describe the behavior of a network. We then proposed two novel approaches, called Atomic Flows and Maximal Flows, to limit or simplify the number of generated flows, in order to respond to scalability issues. Both the two approaches have different characteristics, so advantages and disadvantages. The computed Flows can then be used within automatic verification tools to solve various network-related problems. The type of problem they are going to solve and the specific tool with which they are going to interface will determine which of the two approaches perform best. As future works, we are planning to extend our approach by supporting other types of verification, related for example to information disclosure, latency constraints, and reliability.

References

[1] W. Yang and C. Fung, "A survey on security in network functions virtualization," in *Proc. of the IEEE NetSoft Conference*, 2016.

[2] L. Durante, L. Seno, F. Valenza, and A. Valenzano, "A model for the analysis of security policies in service function chains," in *Proc. of the IEEE Conf. on Network Softwarization (NetSoft)*, 2017.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Tran. on Dep. and Sec. Comp.*, 2022, in press.

[10] E. G. Wong, "Validating network security policies via static analysis of router acl configuration," Tech. Rep., 2006.

[11] 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.

[12] A. R. Khakpour and A. X. Liu, "Quantifying and querying network reachability," in *Proc. of the Inter. Conf. on Distr. Comp. Systems*, 2010.

[13] 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.