POLITECNICO
DI TORINO

Master of Science in Computer Engineering

Master Degree Thesis

Traffic flow and network security function models

Supervisors
prof. Riccardo Sisto
prof. Guido Marchetto
dott. Fulvio Valenza
dott. Daniele Bringhenti

Candidate
Simone Bussa

Academic Year 2020-2021

Summary

With respect to traditional networks, *virtualized networks* introduce some advantages, such as the possibility to use flexible software appliances instead of dedicated hardware devices and the possibility to dynamically reshape and reconfigure the networks themselves. Thanks to such flexibility and dynamicity, they also enable an increased level of network automation, which can be exploited to obtain network solutions that are not only more adaptive to changes, but also less prone to human errors. In this context, nowadays we are assisting at some first research attempts to exploit network virtualization to automate and optimize the allocation and configuration of network security mechanisms. However, these attempts are still quite limited compared to what could be achieved with these paradigms.

One of the aspects that needs further investigation is how *traffic flows and network functions* can be modelled efficiently in order to forecast the behaviour of a network that may be made of different components, including stateful ones. The goal of this thesis is to propose different network modelling approaches, that could be used to solve the problem of automatically defining the allocation and configuration of security mechanisms in a virtualized network.

More precisely, two different (and alternative) models for describing traffic flows and network functions have been proposed and compared. Each model must enable the computation of how a packet that enters the network is forwarded and transformed when crossing the various nodes (i.e., NAT, Load balancer, VPN gateway etc). Such computation, in turn, is necessary to find the optimal placement and configuration of security functions like firewalls, on the basis of given high-level user requirements.

The first approach for describing traffic flows that has been considered makes use of *Atomic Predicates*, a concept recently proposed by some researchers for computing network reachability. This concept has been adapted to our purposes by introducing some new substantial differences, but keeping the basic idea. Given a set of predicates (identified by the IP quintuple), it is possible to compute the set of totally disjunct and minimal predicates (atomic) such that each predicate can be expressed as a disjunction of a subset of them. In other words, it is possible to split each complex predicate (representing for example a firewall rule, a NAT input class, a requirement source, etc) into a set of simpler and minimal atomic predicates.

The second approach, instead, is based on a totally different idea that we call *Maximal Flows*. If with atomic predicates we try to split the traffic flows into smaller atomic flows (reaching the highest level of granularity but also a higher

number of flows), with this second approach we try to do the opposite work, that is to reduce the number of generated flows, aggregating as much as possible different flows into maximal flows representative for all the ones that have been joined. All flows represented by the same maximal flow must behave in the same way when crossing the various nodes of the network, so that it is sufficient to consider the maximal flow and not each single flow that it represents.

Each one of the two described models has its pros and cons, and crucial, besides the formal description of the main algorithms and their implementation, has been the work of comparing performance against scalability testing, for highlighting their difference and feasibility in real scenarios. The proposed implementation in Java, instead, aims to be a contribution and extension to an already existing framework, VEREFOO.

Acknowledgements

To all the supervisors of this thesis. Their help and advice were indispensable. To all those who were alongside me in this academic path that is coming to an end. My sincere thanks go to everyone.

Contents

List of Figures

Chapter 1

Introduction

1.1 Thesis introduction

The advent of *Network Functions Virtualization* (NFV) and *Software-Defined Networking* (SDN) has led to the birth of an increasing number of automated tools based on formal methods. To give some examples, scientific research in recent years has focused on proposing automated and efficient solutions to solve, between the many, two problems, which are connected one to each other: the problem of *Verifying Network Reachability* (e.g., "exists a path that links host a to host b", "a packet with a certain header value can reach host c") and the problem of *Verifying Essential Network Properties*, including security properties that the network must satisfy.

The growing level of automation introduced by NFV and SDN allows to reach greater flexibility and dynamism, avoiding manual work that is more prone to human errors and certainly less reactive to network changes. Furthermore, many times the manual static analysis, which involves designing and configuring the network by hand by a network manager, is not possible due to the size of the network itself, measured in number of nodes and variables to be taken in consideration.

In this context, nowadays, it has become essential to be able to exploit automation especially in the field of Cybersecurity, as suggested in [1] and [2]. Given any network, it should be possible to express a series of *Network Security Requirements* (NSRs) that the network must satisfy, to be defined safe and reliable facing the variety and constant evolution of cybersecurity attacks. In particular, what scientific research is trying to do is to exploit network virtualization to automate and optimize the allocation and configuration of network security mechanisms, such as packet filters ([3], [4], [5]). The main goal of such solutions is to allow the user to specify a series of NSRs, expressing them in a high-level and user-friendly language (e.g., "node a must not communicate with node b", "server x can only be reachable through port y") and then automatically translate these requirements by drawing a graph of the network that is conflict free, with security functions allocated in specific points and configured automatically. The assurance of correctness is provided by formal methods (correctness-by-construction). These solutions are called Refinement tools.

The efforts made to find these solutions exist, but they are not as many as expected and in many cases they are limited to be adopted in simple networks. If on the one hand great progress has been made in literature in developing models with efficient algorithms for verifying networks of packet filters, representing only forwarding tables and access control lists (in which each packet can be either forwarded or dropped but not transformed) - HSA [6], NetPlumber [7], VeriFlow [8] - on the other hand there are not so many solutions for networks that include also packet transformers. Designing an automated tool that also provides support for the use of packet transformers as well as filters presents major challenges. State-of-the-art verification tools fail in those challenges mainly for reason of efficiency and scalability. And this is a great limitation if we consider how many packet transformers are applied in today's networks: NATs that modify the header of the packet, MPLS tunnels that perform label switching, IP-in-IP tunnels used by IPsec, tunnels used for the co-existence of IPv4 and IPv6 that performs header encapsulation and de-encapsulation etc. For this reason, automated tools working on network models that only include packet filters but not packet transformers would not be very useful for most of today's networks.

What is really missing in literature and need further investigation, is how to model traffic flows and network functions, in order to forecast the behaviour of the network. Traffic Flows are used to represent the set of all the possible flows of packets that can cross a network, each traffic flow describes the transformations that affect a certain packet along a certain path: it takes in consideration not only how the packet exits from the source node, but also how it is transformed crossing the various nodes it encounters travelling from source to destination. Some works in this direction are [9], [10] and [11]. The difficulty lies in the fact that, in modern network, it is quite difficult to have a priori a clear vision of what could happen at runtime because many functions perform transformations and are stateful. It is therefore increasingly necessary to find a model that can describe in an efficient way these transformations by means of Traffic Flows. The computation of all possible Traffic Flows for the network, in turn, is necessary to find the optimal placement and configuration of security functions like firewalls, on the basis of given high-level user requirements and so to respond to the Refinement problem described above.

Study, propose and compare different network modelling approaches that could be used to solve the problem of automatically defining the allocation and configuration of security mechanisms in a virtualized network is the main goal of this thesis.

1.2 Thesis description

The remaining of this thesis is organized as follow:

- **Chapter** 2 (BACKGROUND I): describes what is the state-of-the-art in traffic modelling. So, it is a description of the latest novel approaches used to describe traffic packets and how each element of the network can be described as a function that models those packets. In particular, we focus on a novel approach proposed by two researchers Yang and Lam first in 2015 ([12], [13]),

based on what they called Atomic Predicates, that can help in Refinement problems.

- **Chapter** 3 (BACKGROUND II): provides a brief description of the Network Refinement Process and presents VEREFOO (VErified REFinement and Optimized Orchestration) that is an existing tool for Refinement, whose framework this thesis aims to be a contribution and extension of.

- **Chapter** 4: describes the objective of this thesis, introducing the central work done, that is the definition of a new model for representing a class of packets, Predicate, and two novel approaches for defining traffic flows over Predicates.

- **Chapter** 5: shows the definition of a new model for representing a class of packets (how it has been modelled, implemented and which operations are possible over it) that we call Predicate.

- **Chapter** 6: Atomic Flows. This chapter introduces the first novel approach for defining traffic flows that makes use of Atomic Predicates described in Chapter 1. Chapter 5 describes how this concept has been adapted to our purposes by introducing some new substantial differences but keeping the basic idea.

- **Chapter** 7: analyses how the approach with Atomic Flows performs and scales against different scalability test cases.

- **Chapter** 8: Maximal Flows. Introduction to the second novel approach. This Chapter shows what is the basic idea behind this approach and what are its related algorithms.

- **Chapter** 9: analyses the performance of this second approach on the same tests done for Atomic Flows.

- **Chapter** 10: final comparison between the two proposed approaches. Pros and cons of each one and final analysis for determining which could be considered "the best one" for solving the Refinement problem.

- **Chapter** 11: Conclusions

- **Appendix** A: shows a possible Java implementation for the main functions of the Atomic Flows approach.

- **Appendix** B: shows a possible Java implementation for the main functions of the Maximal Flows approach.

Chapter 2

Traffic flows modelling (BACKGROUND I)

A network can be modelled as a graph of nodes. A node can be any function that works within the network (i.e., web client, web server, router, firewall, NAT etc). Each node has a set of input and a set of output ports, each port is controlled by an ACL, that describes whether a packet with a certain header can pass through that port or not, determining what is the forwarding domain of the node: I_a for packets allowed to pass, I_d for those denied. Once entered the node (through the input port), the packet is transferred to the corresponding output port (switching operation), chosen according to the forwarding rules set in the forwarding table of the node, which in turn has been built by routing protocols. Inside the node and before exiting through the output port, the packet could be transformed. As mentioned in the introduction, the most common transformations are header rewriting, encapsulation, de-encapsulation, label switching. Therefore, in addition to the forwarding table and the domains I_a and I_d, in many nodes of the network there is also another table, which is in charge of deciding whether a packet has to be transformed or not, and if yes how it has to be transformed. This transformation behaviour is modelled according to a function T, that has a series of input domains to which corresponds a series of output domains. For a forwarder that simply does forwarding of the packet without modifying it, for example, function T has a single input domain D, that matches all the packets, and T is modelled as the Identity function. For a NAT instead, there are three main domains, D_1, D_2, D_3, which are followed by 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, one that matches with D_3 no transformations are applied to and the packet is simply forwarded.

2.1 Predicates

The choices of forwarding and transformation of a packet are made based on the content of its header. Hence the need to model packet headers as Predicates capable of describing them. The variables of such predicates represent packet header bits

or fields. This means that packets with the same header are represented by the same Predicate and are treated in the same way by all nodes they pass through. Scientific research shows that the choice of how these predicates are modelled and built is trivial and there are an increasing number of solutions (i.e., BDD, Tuple Representation [14], Wildcards Expressions [6], FDD etc.), that we will discuss in the next section of this Chapter. For now, we just need to know that a packet header can be represented by a class called Predicate, which contains the information of its main header fields, and which lends itself well to being the subject of logical operations such as intersection, union and negation.

In this way, it is therefore necessary that also the rules inserted in the ACL of the nodes, as well as those inserted in the forwarding tables and the input domain for T, are represented by Predicates, so that they can be comparable with the Predicates describing the header of incoming packets. In literature, there are lot of algorithms that can transform any ACL/forwarding table/transformation table into a set of Predicates.

Let us see an **example**, that is how a generic ACL can be transformed into a Predicate. This example is taken from [12]. An Access Control List is a list of rules expressed by a condition and an action (which can be ALLOW or DENY), that determines the forwarding behaviour of a packet crossing the node. The order in which these rules are inserted into the list is very important: a packet could match with conditions of several rules, but the action it will undergo is determined by the action of the first rule in sequential order whose condition is satisfied, according to the so called "first match" criterion.

Algorithm 1 Converting an ACL to a Predicate

Input: An ACL (G_i, $action_i$ for $i = 1, ..., m$)
Output: A Predicate for the ACL

1: $allowed \leftarrow false$, $denied \leftarrow false$
2: **for** $i = 1, ..., m$ **do**
3: **if** $action_i = deny$ **then**
4: $denied \leftarrow denied \vee G_i$
5: **else**
6: $allowed \leftarrow allowed \vee (G_i \wedge \neg denied)$
7: **end if**
8: **end for**
9: **return** *allowed*

First, we convert the condition of each rule into a Predicate G_i. Then we run the Algorithm shown in figure, that takes in input the list of rules in sequential order, and transform them into a set of Predicates representing the I_a domain (*allowed* in the code) and a set of Predicates representing the I_d domain (*denied* in the code). The two sets, *allowed* and *denied*, are initially set to false, which in the Set theory represents the empty set. Then for each rule scanned, if the action is DENY, its G condition is simply added in OR to the denied set. Instead, if the action is ALLOW, then its condition is added in OR to the allowed set only after removing

the part of the condition that intersects with denied. This step is essential in order to maintain the logic introduced by the "first match" criterion. At the end, the algorithm returns the set *allowed*, that is the set of Predicates for the I_a domain. The set for the I_d domain can be obtained as the negation of I_a. In other words, *allowed* contains the disjunction of all the Predicates representing the conditions for which the action to perform is ALLOW. Any packet represented by a Predicate in this set, is let pass through the node. We can also note that at the end of the algorithm we obtain a representation of the ACL that completely abandoned the concept of "first match".

Similar operations can be done to covert forwarding tables and input classes for transformers. The goal of these algorithms is always the same, that is the achieve a representation of network function domains as sets of Predicates.

2.2 How can be a Predicate modelled?

The choice for the data structures used to represent Predicates, as mentioned before, is crucial and can affect both space and time efficiency of automated tools that use them. Nowadays, the state-of-the-art has identified several solutions: among these we mainly mention three different models - BDD, Tuple Representation and Wildcards Expressions - but we will analyse in detail only one - BDD -, that is the one used by researchers Yang and Lam as cornerstone for their approach based on Atomic Predicates. We will describe its advantages and provide a practical example, to allow the reader to concretize the concept of Predicate described in the previous sections. However, despite the many advantages of BDDs, it must be immediately said that, for the work of this thesis, was chosen a fourth representation, novel in literature and introduced here by me for the first time. The main reason is that the work of this thesis aims to be a contribution and extension of an existing Refinement tool, *VEREFOO*, implemented in Java. Since there is not an implementation of BDDs in Java, it was necessary to study and propose a new representation that was Java compatible and therefore usable within the *VEREFOO* framework. We will present this new model in Chapter 4.

2.2.1 BDD

BDD stands for Binary Decision Diagram. It is an acyclic, direct and rooted graph structure used to represent Boolean functions. It consists of central nodes (decision nodes) and two terminal nodes labelled as TRUE and FALSE. Each decision node is represented by a Boolean variable and has two children paths, which respectively represent the path to take if the Boolean variable evaluates to true and the one to take if it evaluates to false.

In our case, each variable inside the BDD represents a bit of the packet header. The three graphs in figure 2.1, for example, represent a header field made up of four bits (x_0, x_1, x_2, x_3). A dotted edge denotes an assignment to false, a solid edge denotes an assignment to true. In (a), the BDD subgraph stands for prefix 101* (notice that the variable x_3 is not represented in the graph because it is meaningless

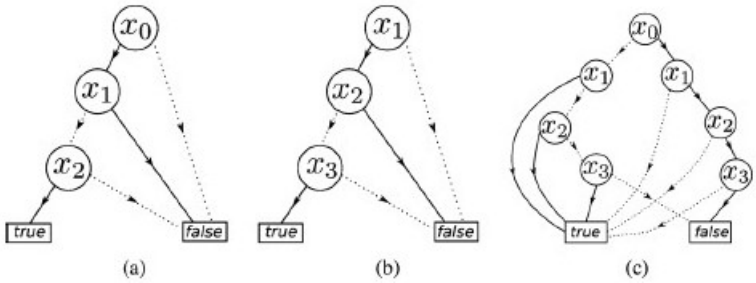


Figure 2.1: BDD subgraphs representing a prefix, a suffix and an interval

for the choice), in (b) it stands for suffix *101, while in (c) stands for the interval from 0001 to 1110.

With this representation, each header field can be represented by a BDD subgraph. The BDD graph representing the whole Predicate is obtained by merging the subgraphs representing the single header fields.

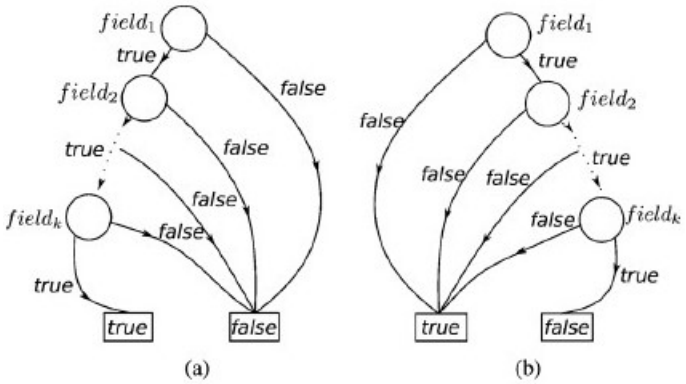


Figure 2.2: BDD representation of an ACL rule. (a) *action* = *allow*, (b) *action* = *deny*

In figure 2.2, each circle represents the BDD subgraph for field-i. An edge exiting the circle is labelled to true if the corresponding subgraph evaluates to true. Otherwise, an edge exiting the circle is labelled to false if the corresponding subgraph evaluates to false. For a Predicate representing an ACL rule, if the rule has action = ALLOW, its BDD graph evaluates to true if all subgraphs evaluate to true. If the rule has action = DENY, its BDD graph evaluates to false if all subgraphs evaluate to true.

BDDs can, in this way, be used to model the network Predicates. In particular, compared to the other approaches mentioned above, they could bring three main advantages:

- **Unique representation.** Considering a Predicate representing a set of packets, its representation as a BDD is unique. Hence, it is not possible for the same Predicate to be represented by two different BDDs. At the contrary, if the representation were not unique, it would have been trivial time consuming to check if different representations thus refer to the same Predicate.

- **Representation size.** It can be shown that the number of nodes inside a BDD used to represent a packet header is $<= 2h + 2$, where h is the number of header bits and $+2$ stands for the two terminal nodes (true and false). All the other representations have greater complexity (in some cases even 2^h).
- **Logical operations.** BDDs lend themselves very well to logical operations, being made up exclusively of true/false Boolean variables. Conjunction and disjunction require time proportional to the product of operand sizes while computing the negation is much easier, it simply consists of swapping the two terminal nodes.

2.3 Atomic Predicates

Although the representation through BDD has proved very effective in representing the Predicates of the network, the complexity in general of these kinds of representation could affect the performance of Refinement tools that continuously perform intersection and union operations over packet sets. Using Predicates directly as the base variable within the Refinement tools could be disadvantageous, regardless of the formal representation used to represent them. And this is the origin of the idea developed by two researchers, Yang and Lam, who in 2015 proposed an alternative approach based on what they called Atomic Predicates.

Given a set of Predicates representative of the network, the two researchers present an algorithm to compute a set of corresponding Atomic Predicates, that is minimal and unique. As stated in [12] *"Atomic predicates are the smallest set of disjunct predicates such that each predicate, of the set over which they are computed, can be expressed as a disjunction of a subset of them"*. In particular,

Definition 2.3.1 *Atomic Predicates: Given a set $\mathcal{P}$ of predicates, its set of Atomic Predicates $\{p_1 \ ... \ p_k\}$ satisfies these five properties:*

1. $p_i \neq$ false, $\forall i \in \{1, ..., k\}$.
2. $\vee_{i=1}^{k} p_i =$ true.
3. $p_i \wedge p_j =$ false, if $i \neq j$
4. Each predicate $P \in \mathcal{P}$, $P \neq$ false, is equal to the disjunction of a subset of atomic predicates

   $P = \bigvee_{i \in S(P)} p_i$, where $S(P) \subseteq \{1, ..., k\}$
5. k is the *minimum* number such that the set $\{p_1, ..., p_k\}$ satisfies the above four properties

The main advantage of Atomic Predicates is that, being unique, they can be built as BDD (all the operations to compute the set of atomic predicates starting from network predicates can be done exploiting the advantages of BDD), and then be identified uniquely by integers. The conjunction (and disjunction) of two

predicates, then, can be computed as the intersection (union) of two sets of integers. Refinement tools, therefore, will have in input simple integers and no longer complex classes.

Yang and Lam, in their papers [12] [13], present a series of algorithms that allow to compute Atomic Predicates starting from a set of Predicates representative of the network. Let us see these algorithms and then a practical example of their application.

Given a Predicate P, the atomic predicates corresponding to $A(\{P\})$ are computed as follow:

$$A(\{P\}) = \begin{cases} \{true\}, & \text{if } P = \text{false or true} \\ \{P, \neg P\}, & \text{otherwise} \end{cases} \tag{2.1}$$

Given two sets of atomic predicates $P_1 = \{b_1, ..., b_l\}$ and $P_2 = \{d_1, ..., d_m\}$, their union $P_3 = A(P_1 \cup P_2) = \{a_1, ..., a_k\}$ is equal to

$$\{a_i = b_{i_1} \wedge d_{i_2} | a_i \neq false, i_1 \in \{1, ..., l\}, i_2 \in \{1, ..., m\}\} \tag{2.2}$$

So, at this point, we use the following algorithm to compute the set of atomic predicates, given a set of predicates P

Algorithm 2 Computing Atomic Predicates

Input: $\{P_1, P_2, ..., P_N\}$
Output: $A(\{P_1, P_2, ..., P_N\})$

1: **for** $i = 1$ to N **do**
2: compute $A(\{P_i\})$ using (2.1)
3: **end for**
4: **for** $i = 2$ to N **do**
5: compute $A(\{P_1, ..., P_i\})$ from $A(\{P_1, ..., P_{i-1}\})$ and $A(\{P_i\})$ using (2.2)
6: **end for**
7: **return** $A(\{P_1, ..., P_N\})$

Each of the atomic predicates in the returned set is then assigned an integer.

Let us take a **practical example** and consider two predicates represented by the IP quintuple q = {IP source, port source, IP destination, p destination, prototype}. Each field of this quintuple represents a field of the packet IP header. We use the wildcard "*", associated with a field, when we want to indicate that there are no limitations for that field (its value may be equal to any possible value within its domain). The five fields of the quintuple are set together in AND.

P_1 = {10.0.0.1, *, *, *, *} represents any IP packet with IP source = 10.0.0.1. There are not limitations for the other fields.

P_2 = {*, *, 10.0.0.2, *, *} instead, represent any IP packet whose destination is 10.0.0.2.

These two predicates can express, for example, a condition in an ACL, an input domain for a transformer or simply a forwarding rule. Also note that they have overlaps: a packet {10.0.0.1, *, 10.0.0.2, *, *} would intersect with both.

The first step is to apply formula 2.1 to P_1 and P_2.

$A(\{P_1\}) = \{10.0.0.1, *, *, *, *\}_{(1)}$, $!\{10.0.0.1, *, *, *, *\}_{(2)}$ (Notice that symbol "!" stands for the negation, so the second predicate means the set of all packets with an IP source different from 10.0.0.1)

$A(\{P_2\}) = \{*, *, 10.0.0.2, *, *\}_{(3)}$, $!\{*, *, 10.0.0.2, *, *\}_{(4)}$

Now we can apply formula 2.2 in order to compute $A(\{P_1, P_2\})$.

$(1) \wedge (3) = \{10.0.0.1, *, 10.0.0.2, *, *\}$

$(1) \wedge (4) = \{10.0.0.1, *, !10.0.0.2, *, *\}$

$(2) \wedge (3) = \{!10.0.0.1, *, 10.0.0.2, *, *\}$

$(2) \wedge (4) = \{!10.0.0.1, *, !10.0.0.2, *, *\}$

The resulting set, $A(\{P_1, P_2\})$, is equal to {{10.0.0.1, *, 10.0.0.2, *, *}, {10.0.0.1, *, !10.0.0.2, *, *}, {!10.0.0.1, *, 10.0.0.2, *, *}, {!10.0.0.1, *, !10.0.0.2, *, *}}.

We have obtained four final atomic predicates, which are unique and disjoint one from each other. We can assign each of them an integer identifier.

{10.0.0.1, *, 10.0.0.2, *, *} = AP1

{10.0.0.1, *, !10.0.0.2, *, *} = AP2

{!10.0.0.1, *, 10.0.0.2, *, *} = AP3

{!10.0.0.1, *, !10.0.0.2, *, *} = AP4

Starting predicates, P_1 and P_2, can be expressed as a disjunction of a subset of the above computed atomic predicates. In particular,

$P1 = AP1 \cup AP2$

$P2 = AP1 \cup AP3$

(Final note for this Chapter: in describing this model, we have defined an example considering IP headers. The same, however, is valid for all protocols and abstract the specific fields of application)

Chapter 3

Refinement Problem (BACKGROUND II)

We are in the context of *Security Automation*. In recent years, there has been much discussion about how automated policy-based network security management tools can assist human in creating and configuring reliable security services capable of verifying and satisfying a series of security requirements given them in input. In addition to the scientific researches mentioned in the introduction of this thesis, we can add [15], [16] and [17]. While the misconfiguration of *Network Security Functions* (NSFs) has recently become the third most critical exploit for cybersecurity attacks, due to the inability of the human being to have a global vision of the entire network that consequently leads to a distribution of filtering rules on NSFs based more on common sense than on a true exhaustive and deterministic analysis, on the other hand the advantage introduced by exploiting security automation can become fundamental to avoid or minimize human errors ([1]). The main task of *Refinement tools* is precisely to refine high-level security requirements by automatically allocating and configuring the necessary security functions. They are approaches predominantly based on formal methods that guarantee correctness-by-construction. These tools can provide for both the design of a complete end-to-end service, also including network functions not related to security aspects, and for the enrichment of an already configured network by introducing security functions built ad hoc for that network. One of the main use cases is that in which there is a service designer who wants to define a Service Graph manually and then enforce it with security functions in an automatic way, because he does not have the security skills to do it by himself.

Among the many interesting aspects of these tools, there is the possibility of seeking optimality: so having all the security functions allocated, configured and conflict free, and at the same time minimize the consumption of resources. One of the problems which arises, instead, is the huge variety of tools that can be used to orchestrate the virtual functions, each one having different peculiarities and characteristics and so being difficult to interface with the others. And this has an impact on the portability of automatic processes that allocate and configure virtual security functions, that must work with a large number of APIs and data formats.

After all these considerations, in the rest of the chapter we will introduce an

existing Refinement framework called *VEREFOO* (VErified REfinement and Optimized Orchestration). The work done for this thesis, as previously mentioned, aims to be a contribution and extension of this tool.

3.1 VEREFOO, general presentation

"*VEREFOO* manages the creation, configuration and orchestration of a complete end-to-end network security service following a modular approach, that is reflected by the design of the framework itself. VEREFOO automatically performs, on a provided Service Graph, an optimized allocation and configuration of the *Network Security Functions* (NSFs) that are necessary to fulfil an input set of *Network Security Requirements* (NSRs), which can be expressed by the service designer by exploiting a high-level language".

The VEREFOO general architecture is shown in figure 3.1. It follows a brief description of its behaviour.

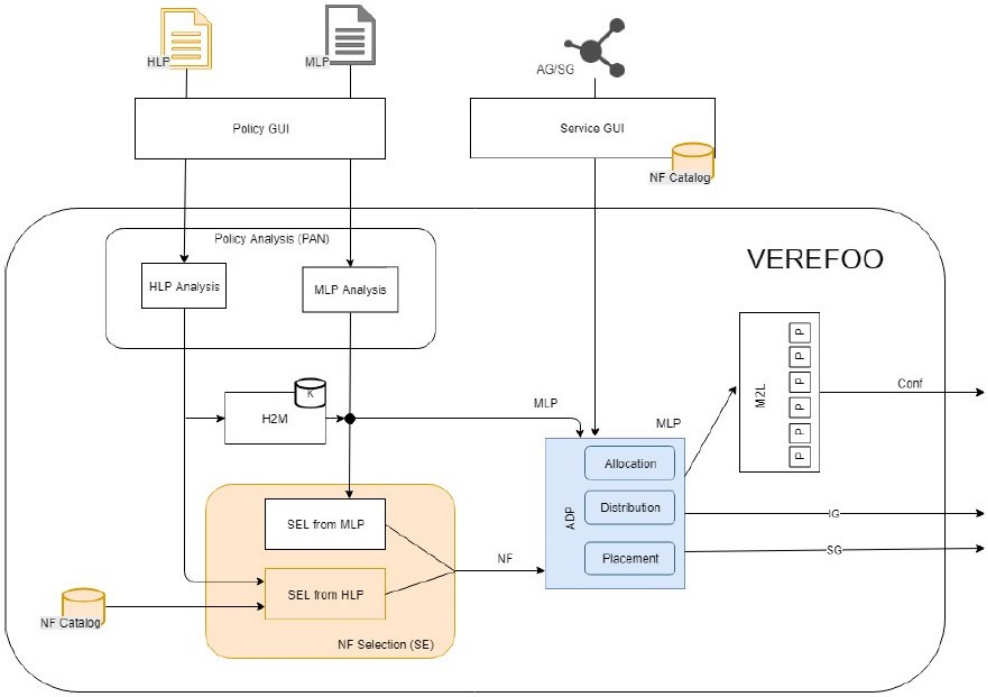


Figure 3.1: VEREFOO General Architecture

- First of all, the user can define the Network Security Requirements through the **Policy GUI**, expressing them as HLP or MLP depending on the experience level of the user. HLP stands for *High-Level Policies* and refers to requirements expressed in a user-friendly language such as "Block all traffic from/to social networks", "Log access to all web sites". MLP instead stands for *Medium-Level Policies* and refers to requirements described with much more level of detail such as "Block all IP traffic coming from 10.0.0.1 having source port equal to 80".

- Requirements expressed in high-level language are automatically translated into medium-level language by the **H2M** (*High-To-Medium*) Module. Medium-level language contains all the useful information necessary for the future creation of the policies of the NSFs.

- A preliminary phase is represented by the *Policy Analysis module* (**PAN**). The goal of this module, which receives the NSRs as input, is to make sure that there are no errors in the NSRs and that they are conflict-free. It returns the minimum set of constraints that must be satisfied or a report highlighting which are the errors in case the conflicts cannot be automatically solved.

- At this point, a fundamental role is played by the *NF Selection Module* (**SE**), which, looking at the expressed NSRs, decides which are the NSFs necessary to satisfy them, choosing them from a pre-built catalogue (**NF catalogue**) which includes all the available functions for the system.

- Finally, the *Allocation, Distribution and Placement Module* (**ADP**) is the central element of the architecture. Its purpose is to compute the final Service Graph with the added NSFs, receiving in input the medium-level NSRs, the list of selected NSFs and the original Service Graph, that describes the topology of the network in turn transformed into the corresponding Allocation Graph (see more later). In order to do this, the ADP module uses a **partial weighted MaxSMT** problem solver, z3Opt. These kinds of tools are an extension of the classic SMT solvers in optimization contexts: the security requirements are introduced into the solver as hard constraints that must be satisfied at all costs, while other specifications can be introduces as weighted and optional soft constraints in order to find the ideal and optimal allocation of the NSFs within the network. The word *"partial"* stands for the fact that there are constraints that are not relaxable and must be satisfied, and others not necessary to solve the problem but introduced only for optimization purposes. The word *"weighted"* instead means that each clause is assigned a weight that contribute to finding the best solution that gives priority to most valued clauses. In terms of computational complexity, the MaxSMT problem is *NP-complete* but, despite this discouraging worst case, with a convenient formulation of the problem and proper pruning techniques the complexity could be reduced to polynomial time. In the next session of this chapter, we will discuss in detail the various types of constraints applied in VEREFOO.

- The last module presented in figure is the *M2L module* (**Medium-to-Low**). This takes the list of medium-level policy rules, returned by the solver, and translates them into low-level language, that depends on the specific implementation of each network function that must be configured.

3.2 VEREFOO, in more details

Let us see and describe VEREFOO in more details.

3.2.1 Service Graph VS Allocation Graph

As already mentioned in several points, the purpose of the tool is to research the optimal allocation scheme of NSFs and configure them so that they are compliant with the provided NSRs. The network security functions the tool allocates are packet filter virtual instances, providing the self-configuration of their rules, by solving a partial weighted MaxSMT problem. All starts from the definition of the requirements and the graph describing the network topology, called Service Graph, which is given in input to the tool.

A **Service Graph (SG)** is an interconnection of service functions and network nodes. The functions are organized in a graph structure, providing the possibility that multiple paths connecting the same source and destination exist. An important thing to consider is that the SG is defined by the network service designer without involving security considerations, but only providing a network service to the user.

The Service Graph provided by the service designer is automatically transformed into an **Allocation Graph (AG)**. Without further specification, a placeholder node, called Allocation Place (AP), is placed on each link that connect two consecutive nodes. These AP represent the potential points where a firewall instance could be inserted by the optimizer engine to reach the optimal allocation scheme.

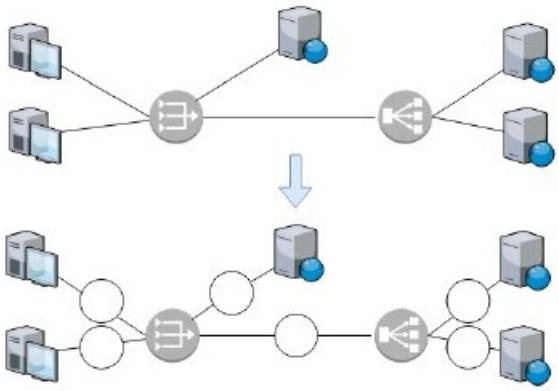


Figure 3.2: Example of Allocation Graph generation. The graph showed above is the Service Graph. The graph below is the Allocation Graph. White circles represent the Allocation Places.

There is one last thing to say. Although the transformation process from the SG to the AG is done automatically, a security service designer can still introduce by hand specific constraints, forcing the allocation of a firewall in a specific position, or prohibiting to consider specific links as potential allocation places. This capability enriches flexibility and, at the same time, decreases computation time by reducing the solution space the optimizer must consider, but can lead to unoptimized solutions.

3.2.2 Network Security Requirements

Among all the security requirements that can be defined, VEREFOO focuses on connectivity requirements between pairs of end points: **Reachability requirements** require that two nodes must be able to communicate, while **Isolation requirements** require they must not communicate. There are four possible approaches used to specify requirements:

1. **Whitelisting approach**: the default behaviour is set to block all possible traffic flows and the user can only additionally specify Reachability requirements.
2. **Blacklisting approach**: the default behaviour is set to allow all possible traffic flows and the user can only specify Isolation requirements.
3. **Rule-oriented specific**: the user can specify both isolation and reachability requirements. The way the system manages all the other cases not specified is automatically decided trying to minimize the number of configured rules.
4. **Security-oriented specific**: the user can specify both isolation and reachability requirements and for the other traffic flows the system allows only the communications that are strictly necessary to satisfy user requirements.

3.2.3 Generating traffic flows phase

A class of packets, also called traffic, is modelled as a Predicate, as we have seen in Chapter 2. Each node in the SG acts on its input traffic and generates a corresponding output traffic. The forwarding and transformation behaviour of the AG is described by means of its set of **traffic flows** F. Each flow f belonging to F, is formally modelled as a list $[n_s, t_{sa}, n_a, t_{ab}, n_b, \ldots, n_k, t_{kd}, n_d]$, that is a list of alternating nodes and traffic. Each list starts from the source node and the corresponfing generated traffic, and then includes all the intermediate nodes, in the path that links source n_s to destination n_d. Each intermediate node can forward the traffic, possibly changing it, or drop it. Traffic t_{ij} represents the traffic transmitted from node n_i to node n_j. All packets represented by t_{ij} are threaten in the same way by node n_j. The set of intermediate nodes includes also the allocation places, with potential firewall configured.

Understanding which traffic flows are generated for each specified requirement is very crucial, in order to know which particular traffic must be blocked (if the requirement is Isolation) or allowed to pass (if the requirement is Reachability), at the generic node n_i.

3.2.4 Description of constraints for the MaxSMT problem

In VEREFOO there are two kinds of constraints: **hard constraints** and **soft constraints**. Hard constraints are not relaxable, they must be satisfied to get a solution of the problem. They are used to model the NSRs and user requirements

about firewall allocation. Soft constraints instead are relaxable and are used to find the optimal firewall allocation and configuration. In particular, VEREFOO considers two main optimization goals: 1) minimize the number of allocated firewall (saving on resource consumption) and 2) minimize the number of rules configured on each allocated firewall (shorter list means filtering operations are done faster and less amount of memory necessary to store it).

In order to describe how constraints are modelled in VEREFOO, let us first introduce some auxiliary functions.

$allocated(n) : N \rightarrow B$. This Boolean function returns true whether a firewall must be allocated in the allocation node n.

$forbidden(l_{ij})$ and $forced(l_{ij}) : L \rightarrow B$. The first function returns true if the allocation of an AP on the link l_{ij} has been prohibited by the service designer, the second one returns true if the allocation has been forced. The two requirements cannot coexist on the same node.

$deny(t) : T \rightarrow B$. It is the function that models the forwarding behaviour of a node. It is true for ingress traffic $t \in T$ if packets represented by t are dropped by the node.

$\pi(f) : F \rightarrow (N)^*$. This function maps a flow $f \in F$ to the ordered list of network nodes that are crossed by that flow, including the destination but not the source.

$\tau(f, n) : F \times N \rightarrow T$. This function maps a flow and a node to the ingress traffic of that node belonging to that flow. In case the flow does not cross the node, this function returns t_0, that means the empty set.

$v(f, n) : F \times N \rightarrow N$. This function maps a flow f and a node n to the next node crossed by f after n. In case n is not in f or it is the last node, the function returns n_0, that means no node.

Hard constraints in firewall allocation

$$\forall l_{ij} \in L.(\neg forbidden(l_{ij}) \implies a_{ij} \in A \wedge l_{ik} \in L \wedge l_{kj} \in L) \tag{3.1}$$

$$\forall l_{ij} \in L.(forbidden(l_{ij}) \implies l_{ij} \in L) \tag{3.2}$$

Function 3.1 means: if on the link between nodes n_i and n_j nothing has been prohibited, then we can insert an allocation place a_{ij}, splitting the original link in two segments l_{ik} and l_{kj}. At the contrary, function 3.2 means: if the creation of the allocation place has been prohibited, link l_{ij} is inserted directly into L without modification.

$$\forall l_{ij} \in L.(forced(l_{ij}) \implies allocated(a_{ij})) \tag{3.3}$$

Function 3.3 is used to force the allocation of a firewall on the link l_{ij}, when the user requests it.

Hard constraints modelling the NSRs

$$\forall f \in F_r.\exists i.(n_i \in \pi(f) \wedge allocated(n_i) \wedge deny_i(\tau(f, n_i))) \tag{3.4}$$

$$\forall f \in F_r.\forall i.(n_i \in \pi(f) \wedge allocated(n_i) \implies \neg deny_i(\tau(f, n_i))) \tag{3.5}$$

Formula 3.4 models an Isolation requirement. The requirement r is satisfied if, for each possible traffic flows computed for r, exists on the path of that flow at least one firewall that is allocated and is configured to block the ingress traffic to that node for that flow. Formula 3.5 instead models a Reachability requirement. The requirement r is satisfied if exists at least a traffic flow computed for r, for which all the nodes in the path of that flow do not block the ingress traffic to that node for that flow.

Soft constraints

Since the solver tries to minimize the number of virtual firewalls allocated, a soft constraint is introduced for each allocation place to express the preference that no firewall should be allocated in that node.

$$\forall a_{ij} \in A.Soft(allocated(a_{ij}) = false, c_k) \tag{3.6}$$

The algorithm tries to maximize the sum of the costs, so putting a high cost c_k to allocated = false means trying to prioritize the fact that the firewall should not be placed there.

For the next constraints, we must introduce another function:

$enforces(d_k, r) : A \times R \to B$. This Boolean function returns true if the default action d_k of a firewall, allocated in the allocation place k, enforces requirement r. For enforces we mean the case in which the default action is aligned to the property of the requirements (i.e., default action = ALLOW and requirement = REACHABILITY or default action = DENY and requirement = ISOLATION). In this case, there is no need to configure any rule inside the firewall for satisfying that requirement.

Starting from this function we can try to minimize the cardinality of the set of rules configured inside each firewall to be allocated, identifying the only security requirements which could actually need a specific rule inside the filtering policies of the firewall. In particular, a firewall could be interested by a specific NSR only if there exists a flow, for that requirement, that cross the allocation place in which the firewall is placed, and if the requirement is not already enforced by the default action of the firewall itself.

$$\forall r \in R.\forall f \in F_r.(a_k \in \pi(f) \wedge \neg enforces(d_k, r) \implies (\forall q \in \tau(f, a_k).q \in Q_k)) \tag{3.7}$$

The set Q_k is the set of all possible tuples (Predicates) describing the traffic incoming to a potential firewall allocated in k, for which a rule is needed. Then, for

each tuple q belonging to Q, a placeholder rule p is created. This placeholder rule has the potential to become a real rule for that firewall, but it is the MaxSMT problem that states if it is necessary or not. Obviously, the action of each placeholder rule is opposite to the default action of the firewall.

$configured(p) : P \rightarrow B$. This formula returns true if the placeholder rule needs to be configured.

A soft constraint is then introduced to try to minimize the number of placeholder rules that must actually be configured, (3.8).

$$\forall p_i \in P_k.Soft(\neg configured(p_i), c_{ki}) \tag{3.8}$$

If at least one rule belonging to P_k must be configured, then necessarily a firewall instance must be allocated in the allocation place a_k. This condition is expressed by the following hard constraint,

$$(\exists p_i \in P_k.configured(p_i)) \implies allocated(a_k) \tag{3.9}$$

Since goal 2) has less priority than goal 1), the value assigned to the weight of the soft clause 3.7 must be greater than the sum of the weights assigned to the soft clauses related to placeholder rules, 3.8. This is expressed by the following formula

$$\sum_{i:p_i \in P_k} (c_{ki}) < c_k \tag{3.10}$$

In general, the cost of allocating a firewall within an allocation place must be greater than the sum of the costs related to all other constraints defined for that allocation place. This choice is critical not only to reach the real objective of the framework but also in terms of performance since, if the solver establishes that a firewall should not be allocated within an allocation place, it does not make sense to try to satisfy all the other policy rule constraints related to that allocation place, because they would be anyway never satisfied.

Fixed that a rule must be configured on a firewall, and therefore that firewall must necessarily be instantiated, then we could try to exploit as much as possible the use of wildcards instead of single values for each field of the rule, in order to aggregate more placeholder rules under a single one. To do this, a weight is assigned to the use of wildcards introducing eleven new soft constraints:

$$\forall p_i \in P_k.\forall j \in \{1,2,3,4\}.Soft(p_i.IPSrc_j = *, c_{ki2j}) \tag{3.11}$$

$$\forall p_i \in P_k.\forall j \in \{1,2,3,4\}.Soft(p_i.IPDst_j = *, c_{ki3j}) \tag{3.12}$$

$$\forall p_i \in P_k.Soft(p_i.pSrc = [0, 65535], c_{ki4}) \tag{3.13}$$

$$\forall p_i \in P_k.Soft(p_i.pDst = [0, 65535], c_{ki5}) \tag{3.14}$$

$$\forall p_i \in P_k.Soft(p_i.tProto = *, c_{ki6}) \tag{3.15}$$

Here too, the weight assigned to the configuration of the placeholder rule must be greater than the sum of weight for using wildcards. In practise, if the placeholder rule is not needed and so not configured, it makes no sense to try to exploit the usage of wildcards for that rule. The formula describing the cost relationship for the three types of soft constraints can be summarized by 3.16

$$\sum_{i:p_i \in P_k} (c_{ki1} + \sum_{j=1}^{4} (c_{ki2j} + c_{ki3j})) < c_k \tag{3.16}$$

and with this we close the discussion on soft clauses.

Other constraints

Finally, there is one more thing to consider in order to finalize the configuration of a firewall consistently with the allocation and configuration choices. It is necessary to define for each allocation place two new hard constraints, taking in account the set of all possible input Predicates.

We must introduce two new formulas:

$matchAll(p, q)$ and $matchNone(p, q) : P \times Q \rightarrow B$. p represents the condition of a rule configured on the firewall; q represents the tuple expressing the Predicate of the input traffic. $MatchAll$ returns true if the rule condition completely includes the traffic tuple (in other words, it means that the intersection between p and q is equal to q). $MatchNone$, on the other hand, is the opposite and means that there is no intersection between the Predicate representing p and the Predicate representing q.

The two new hard constraints are modelled as follow:

$$allocated(a_k) \wedge deny_k(t) \Longrightarrow (a) \vee (b)$$

$$\begin{aligned} &(a) = \text{wlst}(a_k) \wedge \forall q \in t.(\forall p_i \in P_k.(\neg configured(p_i) \vee matchNone(p_i, q))) \\ &(b) = \neg wlst(a_k) \wedge \forall q \in t.(\exists p_i \in P_k.(configured(p_i) \wedge matchAll(p_i, q))) \end{aligned} \tag{3.17}$$

$$allocated(a_k) \wedge \neg deny_k(t) \Longrightarrow (a) \vee (b)$$

$$\begin{aligned} &(a) = \text{wlst}(a_k) \wedge \forall q \in t.(\exists p_i \in P_k.(configured(p_i) \wedge matchAll(p_i, q))) \\ &(b) = \neg wlst(a_k) \wedge \forall q \in t.(\forall p_i \in P_k.(\neg configured(p_i) \vee matchNone(p_i, q))) \end{aligned} \tag{3.18}$$

Formula 3.17 means that, if there is a firewall allocated and it is required to block a traffic t, then

(a) If the firewall is in whitelisting (i.e., default action = DENY), each tuple of the traffic t must not intersect with any placeholder rule (which will have action =

ALLOW), if the placeholder rule is configured, if it is not configured the problem not even arises.

(b) If the firewall is in blacklisting (i.e., default action = ALLOW), then for each traffic tuple there must be at least one configured rule (which will have action = DENY) that does matchAll with that tuple.

Formula 3.18 means that, if there is a firewall allocated and it is asked to let the traffic t pass, then

(a) If the firewall is in whitelisting (i.e., default action = DENY), then there must be at least one configured rule (which will have action = ALLOW) that does full intersection with each traffic tuple

(b) If the firewall is in blacklisting (i.e., default action = ALLOW), then there must be no configured rules (which will have action = DENY) that intersects the traffic tuple.

Chapter 4

Thesis objective

Stated how necessary it is nowadays to be able to exploit the automation that Refinement tools can provide and how important it is, to make these tools work at their best, to arrive at a definition of flexible and efficient traffic flows, in this Chapter we will present the thesis objective. As mentioned in the introduction, one of the aspects that in literature need further investigation is how traffic flows and network functions can be modelled efficiently in order to forecast the behaviour of the network, that may be made of different components, including stateful ones. In Chapter 3, we presented a definition of traffic flows, which make use of the concept of Predicate described in Chapter 2, and which are used by Refinement tools for automatically defining the allocation and configuration of security mechanisms.

Then, we moved on to describe VEREFOO, explaining the problem it must solve and, in general, what is its functioning. All starts with the definition of the requirements by the user. Then, the second step consists of computing the sets of all the possible traffic flows that are affected by at least one requirement. All these classes of packets, that describe the behaviour of the packets crossing the network, are collected in what in this thesis are called Traffic Flows. Then the set of traffic flows are given in input to a Solver that, considering the various nodes of the network and the classes of packet that arrive in input to them, automatically decides where to allocate the security functions and how to configure them, creating the policy necessary to meet the requirements.

Having a clear and flexible modelling for traffic flows, to arrive quickly at a definition of them and then be able to use them efficiently in the context of the Refinement problem, is one of the most important aspects. And with these premises, we can present the thesis objective.

First, the goal of this thesis is to propose a new model able to describe Predicates, explain how they can be built, implemented and which operations are possible over them. Predicates, as we have seen, are the basic element inside a traffic flow. Therefore, having an efficient model to represent them is a good starting point. We will present this new Predicate model in Chapter 5.

Subsequently, we move on presenting and comparing two novel network modelling approaches and algorithms, proposed too for the first time here with this thesis, for defining traffic flows over Predicates. These defined traffic flows can

then be used within the VEREFOO framework or in general within any Refinement tool to solve the Refinement problem. In particular, it will be explained in detail what is the idea that stands behind these two novel approaches, how traffic flows are computed and finally how they interface with the solver internal to VEREFOO. Since each one of the two models has its pros and cons, it has been crucial, besides the implementation, the work of comparing performance against scalability testing, for highlighting their feasibility in real scenarios. In the next paragraph of this Chapter, we will introduce the basic idea that stands behind these two novel approaches. Later they will be treated separately and in more detail in the following Chapter 6 and 8.

4.1 Introduction to two novel approaches for defining traffic flows

Two different (and alternative) models for describing traffic flows and network functions have been identified and compared. Each model must enable the computation of how a packet that enters the network is forwarded and transformed when crossing the various nodes (i.e., NAT, Load balancers, VPN gateways, firewalls etc.).

The first approach that has been considered makes use of Atomic Predicates, a concept described in Chapter 2, proposed in 2015 by some researchers for computing Network Reachability. This concept has been adapted to our purposes by introducing some new substantial differences but keeping the basic idea. Given a set of predicates (identified by the IP quintuple), it is possible to compute the set of totally disjunct and minimal predicates (atomic) such that each predicate can be expressed as a disjunction of a subset of them. In other words, it is possible to split each complex predicate (representing for example a firewall rule, a NAT input class, a requirement source, etc) into a set of simpler and minimal atomic predicates. After having computed the set of atomic predicates for all the "interesting" predicates of the network, we proceed to generate for each user requirement all the related **Atomic Flows**. Atomic flows are flows in which each traffic between two consecutive nodes is an atomic predicate. Then we use these atomic flows as input to the MaxSMT solver, to allocate and configure firewalls with rules whose conditions are expressed by a certain atomic predicate.

The second approach, instead, is based on a totally different idea that we call **Maximal Flows**. If with atomic predicates we try to split the traffic flows into smaller atomic flows (reaching the highest level of granularity but also a higher number of flows), with this second approach we try to do the opposite work, that is to reduce the number of generated flows, aggregating as much as possible different flows into Maximal Flows representative for all the ones that have been joined. All flows represented by the same maximal flow must behave in the same way when crossing the various nodes of the network, so that it is sufficient to consider the maximal flow and not each single flow that it represents. Also in this case, flows are modeled as a list of alternating nodes and Predicates, representing the traffic traveling between two consecutive nodes. Predicates lying in Maximal Flows are no longer necessary Atomic but express the disjunction of several quintuples.

Both proposed approaches lead to the same correct solution of the problem, and for 99% of cases also to the definition of the same flows. In remaining 1% of cases, computed traffic flows differ slightly (depending on if they have been computed with the Atomic or Maximal approach), but obviously lead to the same correct solution of the problem. We will describe later, after introducing more concepts and definitions, some clarifying examples.

Chapter 5

New Predicate Model

This Chapter proposes the definition of a new model to express the predicates of a network. A Predicate is basically a class used to describe a class of packets. For example, if we use Predicates to describe IP packets, this class must contain the useful information included in the IP header of the packet, such as the IP quintuple {source IP, destination IP, source port, destination port, protocol type}. This class can either describe a single type of packet, and in this case its fields will be filled with precise and unique values (e.g., {10.0.0.1, 20.0.0.1, 200, 80, UDP}), or even represent a set of packets, and in this case its fields will be filled with ranges of values or will use wildcards (e.g., {10.0.0.1, 20.0.0.*, 200, [80-100], TCP}).

In Chapter 2, we presented a model used to describe Predicates, Binary Decision Diagram (BDD), and we said that, although they are an efficient and flexible model, there is not a real implementation of them in Java, so it is not easy to use them directly within the framework VEREFOO, which uses Java as programming language. It has been therefore necessary to model a new class for defining Predicates and develop it in Java.

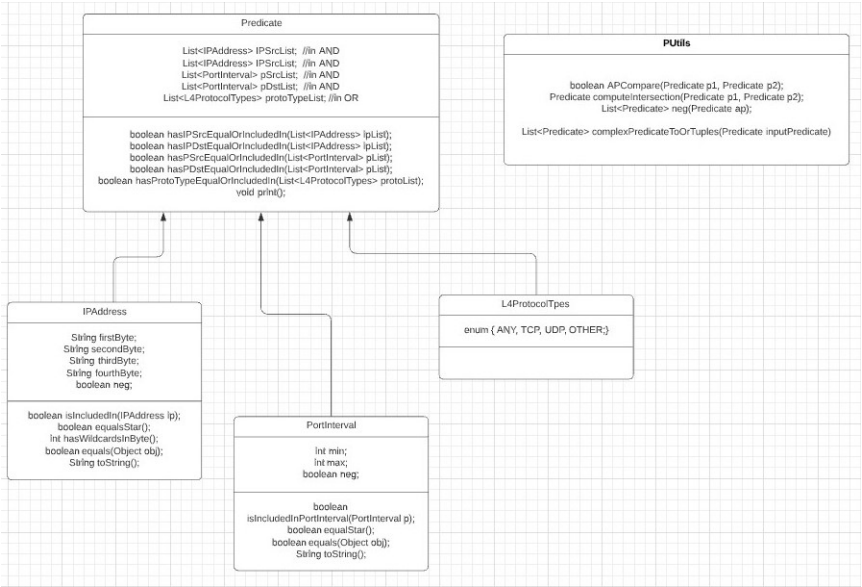


Figure 5.1: UML class diagram describing Predicate

Let us start discussing this class, whose UML diagram in shown in fugure 5.1, following a bottom-up approach.

5.1 IPAddress

Let us start describing the IPAddress class, which is used to model IPv4 addresses. This class has four fields, each one describing a byte of the IP address. Each one of these four fields can have a value between 0 and 255 or can be represented by a wildcard ("*" represents concisely the full range [0, 255]). From now on, for simplicity, we will represent each IP address thorough dotted-decimal notation.

$ip_1.ip_2.ip_3.ip_4$ where the generic ip_i represent the byte in position i inside the IP address.

For example: 130.192.5.* stands for the IPAddress with firstByte=120, secondByte=192, thirdByte=5 while fourthByte is the wildcard "*" meaning all values inside [0, 255]. In other words, this example identifies all the IP addresses matching 130.192.5.0/24. An important thing to notice is the following: if a wildcard is used to describe byte i, then all subsequent bytes to the right of i must also use wildcards. It is not be possible to have an IP address such as, for example, 10.0.*.7.

The Boolean attribute *neg* indicates whether a class should be considered as described by the other four attributes or interpreted as their negated. For example, 130.192.5.1 with the neg attribute set to true indicates any IP address different from 130.192.5.2. 130.192.5.* with the neg attribute set to true, instead, indicates all the IP addresses that do not begin with 130.192.5. This attribute, as we will see later, will be very useful to model the list of IPAddresses included inside Predicate. For simplicity in future notations, we will indicate the presence of the neg attribute equal to true with the logical negation symbol "!". For the two examples described above, they would have been !130.192.5.1 and !130.192.5.*.

Implementation

```
public boolean isIncludedIn(IPAddress ip) {
    if((firstByte.equals(ip.getFirstByte()) || ip.getFirstByte().equals("-1"))
            && (secondByte.equals(ip.getSecondByte()) || ip.getSecondByte().equals("-1"))
            && (thirdByte.equals(ip.getThirdByte()) || ip.getThirdByte().equals("-1"))
            && (fourthByte.equals(ip.getFourthByte()) || ip.getFourthByte().equals("-1")))
        return true;
    return false;
}
```

isIncludedIn returns true if the instance of the IPAddress class calling this method has values equal or included in *ip*. *Ip* should use wildcards.

equalsStar returns true if all the bytes within the IPAddress are equal to the wildcard.

equalField instead returns true if the two IPAddresses have the same values for all bytes (here we do not consider the value of attribute neg).

```
public boolean equalsStar() {
    if(firstByte.equals("-1") && secondByte.equals("-1") &&
            thirdByte.equals("-1") && fourthByte.equals("-1"))
        return true;
    return false;
}
```

```
public boolean equalFileds(IPAddress ip) {
    if(firstByte.equals(ip.getFirstByte()) && secondByte.equals(ip.getSecondByte()) &&
            thirdByte.equals(ip.getThirdByte()) && fourthByte.equals(ip.getFourthByte()))
        return true;
    return false;
}
```

5.2 PortInterval

This class is used to model source and destination port numbers within an IP header. It can represent a range of values included between min and max, within the range of all possible values starting from 0 to 65535, or a single specific port number (in this case min = max). Here too, it is possible to use wildcards, representing the full range [0, 65535], by setting min = max = -1. As for IPAddress, fundamental is the Boolean attribute neg which indicates whether the PortInterval instance should be interpreted as it is or as its negation.

Examples:
min=10, max=20, neg=false indicates the range [10,20]
min=10, max=20, neg=true indicates the disjunction of [0,9] U [21, 65535]
min=5, max=5, neg=false indicates port number 5
min=5, max=5, neg=true indicates any port number different from 5 (that is, the disjunction [0,4] U [6, 65535])
min=-1, max=-1, neg=false indicates the full range [0, 65535]
min=-1, max=-1, neg=true actually indicates no port numbers and therefore the empty set

Implementation

```
public boolean isIncludedInPortInterval(PortInterval p) {
    if((min >= p.getMin() || p.getMin() == -1) && (max <= p.getMax() || p.getMax() == -1))
        return true;
    return false;
}
```

isIncludedInPortInterval returns true is the instance of PortInterval has values included within *p*.

equalStar returns true if the PortInterval represents the full set of ports [0, 65535]

```
public boolean equalStar() {
    if(min == -1 && max == -1)
        return true;
    return false;
}
```

```
public boolean equalFileds(PortInterval pi) {
    if(min == pi.getMin() && max == pi.getMax())
        return true;
    return false;
}
```

equalFields instead returns true if the two PortInterval represent the same range (here we do not consider the value of attribute neg).

5.3 L4ProtocolType

L4ProtocolTypes class is an enum that contains all the possible values for the IP prototype field. VEREFOO mainly considers two values for protocol type, which are TCP and UDP. All other possible values are represented by the value OTHER. ANY instead represents the wildcard, meaning in this case the disjunction of {TCP,UDP,OTHER}.

5.4 Predicate

Let us now see the definition of the most important class, Predicate, used to simultaneously model the five main fields of the IP header, that are represented by the quintuple {source IP, destination IP, source port, destination port, protocol type}. These five fields are modelled as list of IPAddress, PortInterval and L4ProtocolTypes.

IPAddress List

Let us start from the two lists of IPAddress representing IP source and IP destination. Each element inside a list is in conjunction (AND) with the other elements. We can indicate two main cases:

1. The list contains only one element: in this case the only element corresponds to the representation of the source or destination IP (it depends on which list we are considering, IPSrcList or IPDstList).

   Examples:
   IPSrcList = [10.0.0.1] means that the source of the packet is the IP address 10.0.0.1

IPDstList = [10.0.0.*] means that the destination of the packet can be any IP address matching 10.0.0.0/24

2. The list contains more than one element: in this case the elements are considered in AND one to each other. It is important that the following two properties are satisfied: 1) in case there are more IPAddress elements in the list, there must be only one with neg = false (while all the others must have neg = true) and 2) the one with neg = false must contain at least one wildcard in order to include all the others with neg = true (and not vice versa).

   Examples:
   [10.0.0.*, !10.0.0.1, !10.0.0.2]. This list of IPAddress is used to model all IP addresses that matches 10.0.0.0/24 different from 10.0.0.1 and 10.0.0.2. The fact that the single IPAddresses are placed in conjunction (AND) implies that the formulation of this list is equal to 10.0.0.* AND !10.0.0.1 AND !10.0.0.2. We can see that there is only one element with neg = false (10.0.0.*), which has a wildcard on its fourth byte to include the other IPAddresses with neg = true (10.0.0.1, 10.0.0.2).
   [!10.0.0.1, !10.0.0.2]. This is a second possible example in which there are not IPAddresses with neg = false. In this case, the address with all wildcards *.*.*.* is taken by default and put in conjunction with the other elements. The corresponding logic formulation would be *.*.*.* AND !10.0.0.1 AND !10.0.0.2, indicating any IP address different from 10.0.0.1 and 10.0.0.2.
   All the other cases are not valid, for example:
   [10.0.0.1, 10.0.0.2]. It contains two IPAddress with neg = false. It has no meaning because the conjunction between them returns the empty set. In fact, 10.0.0.1 AND 10.0.0.2 = empty set.
   [10.0.0.1, !10.0.0.*]. In this case there is one element with neg = false and one element with neg = true. The first property is then satisfied but not the second one. In fact, the IPAddress with neg = false (10.0.0.1) does not include the IPAddress with neg = true (10.0.0.*) but it happens the opposite. The corresponding logic formulation is 10.0.0.1 AND !10.0.0.* that is equivalent to the empty set.
   [10.0.0.*, !20.0.0.1]. Here, we have an IPAddress with neg = false and one with neg = true, but the one with neg = false does not include the one with neg = true and it does not happen even the vice versa. This formulation by itself is not an error as it corresponds to 10.0.0.* AND !20.0.0.1 = 10.0.0.*. It is only superfluous to indicate the IPAddress with neg = true. The same formulation could be obtained with [10.0.0.*] (which is a list with a single element).

The main reason for this choice, that is maintaining the Boolean attribute neg within IPAddress and having the elements inside the list in conjunction and satisfying the described above properties, is scalability. Modelling the list as a set of IPAddress in disjunction (OR), without in this case the need of using the neg attribute anymore, would have been much more expensive in terms of memory allocation and time to perform the operations (as the sets would have been much larger).

Examples:
[10.0.0.*, !10.0.0.2, !10.0.0.3]. This list contains 3 elements using the attribute neg and in conjunction one to each other. The corresponding formulation with elements in OR and without the attribute neg, could no longer use the wildcard and it would become [10.0.0.1, 10.0.0.4, 10.0.0.5 ..., 10.0.0.255], a list containing 254 elements (versus 3 of previous representation).
[10.*.*.*, !10.0.0.2]. This would become an even larger list, equal to [10.1.*.*, 10.2.*.*, ... , 10.255.*.*] + [10.0.1.*, 10.0.2.*, ..., 10.0.255.*] + [10.0.0.1, 10.0.0.3, 10.0.0.4 ..., 10.0.0.255], thus containing 255+255+254 = 764 elements (versus 2 of the previous representation).

PortInterval List

Trying to exploit wildcards and the attribute neg as much as possible is essential. This is why also the two list of PortInterval (pSrcList and pDstList) are modelled inside Predicate in the same way. So, here too we make use of the attribute neg and build the lists with element in conjunction one to each other. Within the list, there can only be one element with neg = false which must use wildcards to include any other element inserted with neg = true.

Examples:
[**(10,20)**]. The list contains one single element and represents the range of ports [10, 20].
[**!(10,20)**]. The list contains one single element and represents the set of ports not included in the range [10,20]. This formulation corresponds to [0,9]+[21, 65535].
[**(1, 20), !(4,6)**]**.** This case represents all ports included between the range [1,20] except for those included in the range [4,6]. The corresponding formulation is [1,3]+[7,20]
[**(1, 50), !(10,20), !(30,40)**]. In this case the corresponding formulation is [1,9]+[21,29]+[41,50].
[**(-1, -1), !(5,5)**]. This last one represents all possible port numbers except for 5. The corresponding formulation is [0,4]+[6,65535].

L4ProtocolTypes List

The situation is different for L4ProtocolTypes list. As mentioned in the previous section, this class is an enum which contains four possible values (UDP, TCP, OTHER, ANY). For simplicity, the single elements inserted inside the list are placed in OR one to each other, since the maximum cardinality is very small. With a set of up to two elements, it is very easy to do intersection, union and negation without the need of introducing the concept of neg.

Examples:
protoTypeList = [UDP, TCP] means that the protocol type value of the Predicate can be either UDP or TCP.
protoTypeList = [ANY] means that the protocol type value of the Predicate can be either UDP or TCP or OTHER

5.4.1 Operations on Predicate (implementation)

Now, let us see what are the operations that can be done on Predicates. In addition to the methods included in the Java class, which are auto-explicative and we leave their description in the Appendix of this thesis, those included in the interface PUtils are very important. Among these, there are the methods allowing to perform logical operations between Predicates, such as equivalence, intersection and negation.

APCompare

```
public boolean APCompare(Predicate p1, Predicate p2) {
    //comparing lists size
    if(p1.getIPSrcList().size() != p2.getIPSrcList().size()
            || p1.getIPDstList().size() != p2.getIPDstList().size()
            || p1.getpSrcList().size() != p2.getpSrcList().size()
            || p1.getpDstList().size() != p2.getpDstList().size())
        return false;
    //NOTE: since list1 and list2 have the same size and they don't contain duplicates,
    //list1.containsAll(list2) is sufficient
    //No need to call also list2.containsAll(list1)
    if(!p1.getIPSrcList().containsAll(p2.getIPSrcList()))
        return false;
    if(!p1.getIPDstList().containsAll(p2.getIPDstList()))
        return false;
    if(!APComparePortList(p1.getpSrcList(), p2.getpSrcList()))
        return false;
    if(!APComparePortList(p1.getpDstList(), p2.getpDstList()))
        return false;
    return APComparePrototypeList(p1.getProtoTypeList(), p2.getProtoTypeList());
}

public boolean APComparePrototypeList(List<L4ProtocolTypes> list1,
        List<L4ProtocolTypes> list2) {
    if(!list1.containsAll(list2))
        return false;
    return true;
}

public boolean APComparePortList(List<PortInterval> list1,
        List<PortInterval> list2) {
    if(!list1.containsAll(list2))
        return false;
    return true;
}

public boolean isPredicateContainedIn(Predicate p, List<Predicate> list) {
    for(Predicate p2: list) {
        if(APCompare(p, p2))
            return true;
    }
    return false;
}
```

This function receives two Predicates as parameters in input and returns true if the two are equal, false otherwise. To be equal, two Predicates must have lists containing the same identical elements. So, a first check verifies that these lists have the same size, otherwise we can say Predicates are different. After this check, taking also into account that lists cannot contain duplicate elements, to ensure

equality it is sufficient to check that one list contains all the elements present in the other one (it is not necessary to check also the vice versa). This operation is done for all the five lists (IPSrcList, IPDstList, pSrcList, pDstList, protoTypeList).

computeIntersection

```
public List<IPAddress> intersectionIPAddressNew(IPAddress ip1, IPAddress ip2){
    List<IPAddress> retList = new ArrayList<>();
    if(!ip1.isNeg() && !ip2.isNeg()) { //both not neg
        if(ip1.isIncludedIn(ip2)) {
            retList.add(ip1);
            return retList;
        }
        if(ip2.isIncludedIn(ip1)) {
            retList.add(ip2);
            return retList;
        }
    } else if(!ip1.isNeg() && ip2.isNeg()) { //ip1 not neg, ip2 neg
        if(!ip1.equalFileds(ip2) && ip2.isIncludedIn(ip1)) {
            retList.add(ip1);
            retList.add(ip2);
            return retList;
        }
        if(!ip1.equalFileds(ip2) && !ip1.isIncludedIn(ip2)) {
            retList.add(ip1);
            return retList;
        }
    } else if(ip1.isNeg() && !ip2.isNeg()) { //ip1 neg, ip2 not neg
        if(!ip1.equalFileds(ip2) && ip1.isIncludedIn(ip2)) {
            retList.add(ip1);
            retList.add(ip2);
            return retList;
        }
        if(!ip1.equalFileds(ip2) && !ip2.isIncludedIn(ip1)) {
            retList.add(ip2);
            return retList;
        }
    } else { //both neg
        if(ip1.equalFileds(ip2)) {
            retList.add(ip1);
            return retList;
        } else {
            if(ip1.isIncludedIn(ip2)) {
                retList.add(ip2);
                return retList;
            }
            if(ip2.isIncludedIn(ip1)) {
                retList.add(ip1);
                return retList;
            }
            retList.add(ip1);
            retList.add(ip2);
            return retList;
        }
    }

    return retList;
}
```

A little more complicated, however, is computing intersection. The function must perform the intersection between all the fields present within Predicate, placed in conjunction one to each other. In practise, it is sufficient that one of the five fields

(for example the IP source field) does not have intersection with the corresponding field of the other Predicate, to state that there is no intersection between the two Predicates (intersection equal to the empty set).

In fact, $(IPSrc_1 \wedge IPDst_1 \wedge pSrc_1 \wedge pDst_1 \wedge proto_1) \wedge (IPSrc_2 \wedge IPDst_2 \wedge pSrc_2 \wedge pDst_2 \wedge proto_2) = (IPSrc_1 \wedge IPSrc_2) \wedge (IPDst_1 \wedge IPDst_2) \wedge (pSrc_1 \wedge pSrc_2) \wedge (pDst_1 \wedge pDst_2) \wedge (proto_1 \wedge proto_2)$

Let us first see the code that describes the intersection between single fields.

The function must perform the intersection between two IPAddress. If the intersection exists, the corresponding list of IPAddress (as usual with elements in AND one to each other and with the porperties described above) is returned, otherwise an empty list is returned.

There are mainly four cases to consider:

1. Both ip_1 and ip_2 have neg attribute set to false.

   - If ip_1 is included in ip_2, then it is returned the list containing only ip_1.
     Example:
     ip_1 = 10.0.0.1, ip_2 = 10.0.0.*, ip_1 AND ip_2 = 10.0.0.1 AND 10.0.0.* = 10.0.0.1 = ip_1 so it is returned the list [10.0.0.1].
   - Vice versa if ip_2 is included in ip_1, then it is returned the list containing only ip_2, for the same reason described in the precious point.
   - In all other cases, when ip_1 and ip_2 are disjointed, it is returned the empty set.

     NOTE: function *isIncludedIn* returns true also in the case ip_1 is equal to ip_2.

2. ip_1 has neg = false, ip_2 has neg = true. In this case:

   - If ip_1 and ip_2 have different byte values and ip_2 is included in ip_1, then it is returned the list containing both ip_1 and ip_2.
     Example:
     ip_1 = 10.0.0.*, ip_2 = !10.0.0.1, ip_1 AND ip_2 = 10.0.0.* AND !10.0.0.1 that corresponds to the list [10.0.0.*, !10.0.0.1].
   - If ip_1 and ip_2 have different byte values and ip_2 is not included in ip_1, then it is returned only the list containing ip_1.
     Example:
     ip_1 = 10.0.0.1, ip_2 = !10.0.0.2, ip_1 AND ip_2 = 10.0.0.1 AND !10.0.0.2 = 10.0.0.1 = ip_1, that corresponds to the list [10.0.0.1].
   - All the other cases return the empty set.
     Example when ip_1 and ip_2 have the same byte values: ip_1 = 10.0.0.1, ip_2 = !10.0.0.1, ip_1 AND ip_2 = 10.0.0.1 AND !10.0.0.1 = empty set.
     Example when ip_1 is included in ip_2:
     ip_1 = 10.0.0.1, ip_2 = !10.0.0.*, ip_1 AND ip_2 = 10.0.0.1 AND !10.0.0.* = empty set.

3. ip_1 has neg = true, ip_2 has neg = false. Same for the previous point.

4. Both ip_1 and ip_2 have neg attribute set to true.

   - If ip_1 and ip_2 have te same byte values, it means they are identical and so it is returned a list of a single element equal to their value.
     Example:
     ip_1 = !10.0.0.1, ip_2 = !10.0.0.1, ip_1 AND ip_2 = !10.0.0.1 AND !10.0.0.1 = !10.0.0.1 that corresponds to the list [!10.0.0.1]
   - If ip_1 and ip_2 have different byte values and ip_1 is included in ip_2, then it is returned a list containing ip_2.
     Example:
     ip_1 = !10.0.0.1, ip_2 = !10.0.0.*, ip_1 AND ip_2 = !10.0.0.1 AND !10.0.0.* = !10.0.0.* = ip_2, that corresponds to list [!10.0.0.*]
   - If ip_1 and ip_2 have different byte values and ip_2 is included in ip_1, then it is returned a list containing ip_1.
     Example:
     ip_1 = !10.0.0.*, ip_2 = !10.0.0.1, ip_1 AND ip_2 = !10.0.0.* AND !10.0.0.1 = !10.0.0.* = ip_1, that corresponds to list [!10.0.0.*].
   - If ip_1 and ip_2 have different byte values and none includes the other, then it is returned the list containing both of them.
     Example:
     ip_1 = !10.0.0.1, ip_2 = !10.0.0.2, ip_1 AND ip_2 = !10.0.0.1 AND !10.0.0.2, that corresponds to the list [!10.0.0.1, !10.0.0.2].

The same is done for intersectionPortIntervalNew.

These described two functions are used to compute the intersection starting from two single IPAddress or PortInterval. But, as we said before, the Predicate class does not contain single instances of these classes, but it contains lists of them with elements in conjuction. The intersection then becomes a little more complex and it is computed with the following algorithm.

```
public Predicate computeIntersection(Predicate p1, Predicate p2){
    //Check IPSrc
    List<IPAddress> resultIPSrcList = p2.getIPSrcList();
    List<IPAddress> tmpList;
    List<IPAddress> tmpList2 = new ArrayList<>();
    List<IPAddress> toInsert1List = new ArrayList<>();
    boolean toInsert1;
    for(IPAddress src1: p1.getIPSrcList()) {
        toInsert1 = false;
        for(IPAddress src2: resultIPSrcList) {
            tmpList = intersectionIPAddressNew(src1, src2);
            if(tmpList.isEmpty())
                return null; //no intersection exists
            for(IPAddress res: tmpList) {
                if(res.equals(src1))
                    toInsert1 = true;
                else tmpList2.add(res);
            }
        }
```

```
    if(resultIPSrcList.isEmpty()) toInsert1 = true;
    resultIPSrcList = new ArrayList<>(tmpList2);
    tmpList2 = new ArrayList<>();
    if(toInsert1) toInsert1List.add(src1);
}
resultIPSrcList.addAll(toInsert1List);

//Check IPDst
List<IPAddress> resultIPDstList = p2.getIPDstList();
toInsert1List = new ArrayList<>();
for(IPAddress dst1: p1.getIPDstList()) {
    toInsert1 = false;
    for(IPAddress dst2: resultIPDstList) {
        tmpList = intersectionIPAddressNew(dst1, dst2);
        if(tmpList.isEmpty())
            return null; //no  intersection exists
        for(IPAddress res: tmpList) {
            if(res.equals(dst1))
                toInsert1 = true;
            else tmpList2.add(res);
        }
    }
    if(resultIPDstList.isEmpty()) toInsert1 = true;
    resultIPDstList = new ArrayList<>(tmpList2);
    tmpList2 = new ArrayList<>();
    if(toInsert1) toInsert1List.add(dst1);
}
resultIPDstList.addAll(toInsert1List);

//Check pSrc
List<PortInterval> resultPSrcList = p2.getpSrcList();
List<PortInterval> tmpPList;
List<PortInterval> tmpPList2 = new ArrayList<>();
List<PortInterval> toInsert1PList = new ArrayList<>();
for(PortInterval psrc1: p1.getpSrcList()) {
    toInsert1 = false;
    for(PortInterval psrc2: resultPSrcList) {
        tmpPList = intersectionPortIntervalNew(psrc1, psrc2);
        if(tmpPList.isEmpty())
            return null; //no  intersection exists
        for(PortInterval res: tmpPList) {
            if(res.equals(psrc1))
                toInsert1 = true;
            else tmpPList2.add(res);
        }
    }
    if(resultPSrcList.isEmpty()) toInsert1 = true;
    resultPSrcList = new ArrayList<>(tmpPList2);
    tmpPList2 = new ArrayList<>();
    if(toInsert1) toInsert1PList.add(psrc1);
}
resultPSrcList.addAll(toInsert1PList);



//Check pDst
List<PortInterval> resultPDstList = p2.getpDstList();
toInsert1PList = new ArrayList<>();
for(PortInterval pdst1: p1.getpDstList()) {
    toInsert1 = false;
    for(PortInterval pdst2: resultPDstList) {
        tmpPList = intersectionPortIntervalNew(pdst1, pdst2);
        if(tmpPList.isEmpty())
            return null; //no  intersection exists
        for(PortInterval res: tmpPList) {
```

```
                if(res.equals(pdst1))
                    toInsert1 = true;
                else tmpPList2.add(res);
            }
        }
        if(resultPDstList.isEmpty()) toInsert1 = true;
        resultPDstList = new ArrayList<>(tmpPList2);
        tmpPList2 = new ArrayList<>();
        if(toInsert1) toInsert1PList.add(pdst1);
    }
    resultPDstList.addAll(toInsert1PList);

    //Check proto
    List<L4ProtocolTypes> resultProtoList = new ArrayList<>();
    if(p1.getProtoTypeList().contains(L4ProtocolTypes.ANY))
        resultProtoList =  p2.getProtoTypeList();
    else if(p2.getProtoTypeList().contains(L4ProtocolTypes.ANY))
        resultProtoList =  p1.getProtoTypeList();
    else { //None contains ANY, so compute intersection
        for(L4ProtocolTypes proto1: p1.getProtoTypeList()) {
            if(p2.getProtoTypeList().contains(proto1))
                resultProtoList.add(proto1);
        }
    }
    if(resultProtoList.isEmpty())
        return null; //no intersection exists

    Predicate resultPredicate = new Predicate();
    resultPredicate.setIPSrcList(resultIPSrcList);
    resultPredicate.setIPDstList(resultIPDstList);
    resultPredicate.setpSrcList(resultPSrcList);
    resultPredicate.setpDstList(resultPDstList);
    resultPredicate.setProtoTypeList(resultProtoList);
    return resultPredicate;
}
```

Each list within a Predicate is considered individually and must be placed in intersection with the corresponding list of the other Predicate. Let us describe first how it works the algorithm for the intersection of IPAddress lists related to IP sources. Let us consider the algorithm step by step with an example to make it easier to understand.

Example:
P1 contains internally the IPSrcList = [10.0.0.*, !10.0.0.1]
P2 contains internally the IPSrcList = [!10.0.0.1, !10.0.0.2, !20.0.0.1]
By eye, it is easier to see that the resulting intersection is [10.0.0.*, !10.0.0.1, !10.0.0.2].

The algorithm to arrive at this solution contains a double nested for loop: the outer one that cycles on the elements of the list of P1 (we call this list list1), the inner one that cycles on a list (resultIPSrcList, that we call list2) modified at each iteration of the outer loop and which initially contains all the elements of the list of P2. What we try to do at each iteration is to reduce the cardinality of the two lists, eliminating those elements that are superfluous for the representation of the intersection (in our example, we have to eliminate the IPAddress "!20.0.0.1").

For the outer cycle, we consider one element of list1 at a time, and this element (src1) is placed in intersection with each one of the elements of resultIPSrcList. They are two single IPAddresses, so we can call the intersectionIPAddressNew

method. If this method returns the empty list, it means that the two elements do not intersect and so we can directly say that the intersection of the two whole Predicates is equal to null, since, because of the two lists contain element in conjunction (AND), the intersection of each element of one must exist with all the elements of the other. On the contrary, if returned list is not empty, the code will cycle on the resulting IPAddresses returned. If a resulting element is an IPAddress coming from list1, then we set the Boolean variable toInsert1 equal to true and later (at the end of the outer loop iteration) we will add that IPAddress inside toInsertList1, which is the list that contains IPAddresses coming from list1 that are going to be part of the solution, otherwise if it is an IPAddress coming from list2, we insert it in a temporary list tmpList2, which at the end of outer loop iteration will become the new resultIPSrsList, on which we will cycle in the next internal for. For the example we are considering, after the first external iteration, toInsert1 = [10.0.0.*], resultIPSrcList = [!10.0.0.1, !10.0.0.2]. We note that the IPAddress "!20.0.0.1" has been removed because 10.0.0.* AND !20.0.0.1 = 10.0.0.*, so it is not added to resultIPSrcList. At the end of the second and last external iteration toInsert1 = [10.0.0.*, !10.0.0.1], resultIPSrcList = [!10.0.0.2]. The final list representing the intersection of IPSrcList from P1 with IPSrcList from P2 is given by concatenating toInsert1 with the remaining values contained inside resultIPSrcList after the last iteration.

As we can see from the code, the same operations are applied to compute the intersection of IPDstLists, pSrcLists and pDstLists. The code differs only for the intersection of protoTypesLists, that contain few elements in disjunction. Computing the intersection of these lists simply consists of checking if one of the two contains the element "ANY", in this case the result of the intersection is given by the other list. If instead neither of the two lists contains "ANY", then the code cycles on each element of one list at a time, checking if its value is contained also in the other list. If the value is contained, then we can add it to the final set representing the intersection, otherwise we can discard it. At the end of the double nested loop, if the resulting final set is empty, then it means that the intersection between the two protocolTypesList does not exist and so between the two Predicates.

neg

```
public List<Predicate> neg(Predicate ap){
    List<Predicate> neg = new ArrayList<>();
        //check IPSrc
        for(IPAddress src: ap.getIPSrcList()) {
            if(!src.equalsStar()) {
                Predicate sp = new Predicate(src.toString(), src.isNeg(), "*",
                        false, "*", false, "*", false, L4ProtocolTypes.ANY);
                neg.add(sp);
                Predicate sp2 = new Predicate(src.toString(), !src.isNeg(), "*",
                        false, "*", false, "*", false, L4ProtocolTypes.ANY);
                neg.add(sp2);
            }
        }
```

```
        //check IPDst
        for(IPAddress dst: ap.getIPDstList()) {
            if(!dst.equalsStar()) {
                Predicate sp = new Predicate("*", false, dst.toString(), dst.isNeg(),
                        "*", false, "*", false, L4ProtocolTypes.ANY);
                neg.add(sp);
                Predicate sp2 = new Predicate("*", false, dst.toString(), !dst.isNeg(),
                        "*", false, "*", false, L4ProtocolTypes.ANY);
                neg.add(sp2);
            }
        }
        //check pSrc
        for(PortInterval psrc: ap.getpSrcList()) {
            if(!psrc.equalStar()) {
                Predicate sp = new Predicate("*", false, "*", false, psrc.toString(),
                        psrc.isNeg(), "*", false, L4ProtocolTypes.ANY);
                neg.add(sp);
                Predicate sp2 = new Predicate("*", false, "*", false, psrc.toString(),
                        !psrc.isNeg(), "*", false, L4ProtocolTypes.ANY);
                neg.add(sp2);
            }
        }
        //check pDst
        for(PortInterval pdst: ap.getpDstList()) {
            if(!pdst.equalStar()) {
                Predicate sp = new Predicate("*", false, "*", false,  "*", false,
                        pdst.toString(), pdst.isNeg(), L4ProtocolTypes.ANY);
                neg.add(sp);
                Predicate sp2 = new Predicate("*", false, "*", false,  "*", false,
                        pdst.toString(), !pdst.isNeg(), L4ProtocolTypes.ANY);
                neg.add(sp2);
            }
        }
        //check protoType
        List<L4ProtocolTypes> list = ap.getProtoTypeList();
        //If different from ANY and different from the full set
        if(!list.contains(L4ProtocolTypes.ANY) && list.size() !=
                L4ProtocolTypes.values().length-1) {
            Predicate sp1 = new Predicate("*", false, "*", false,  "*", false,
                    "*", false, L4ProtocolTypes.ANY);
            sp1.setProtoTypeList(list);
            neg.add(sp1);
            List<L4ProtocolTypes> negList = computeDifferenceL4ProtocolTypes(list);
            if(!negList.isEmpty()) {
                Predicate sp2 = new Predicate("*", false, "*", false,  "*", false,
                    "*", false, L4ProtocolTypes.ANY);
                sp2.setProtoTypeList(negList);
                neg.add(sp2);
            }
        }

        return neg;
}
```

Let us now see how it works the function that, given a Predicate, returns the list of Predicates in disjunction (OR) representing the negation of the given Predicate.

Here the code is simpler, given a Predicate having all its fields in AND, its negation is expressed by the disjunction of more Predicates, as stated by the De Morgan Law:

$$\neg(a \wedge b) = \neg a \vee \neg b \tag{5.1}$$

Since the five fields inside Predicate {IPSrcList, IPDstList, pSrcList, pDstList, protoTypeList} are in AND one to each other and each element inside those lists are

in turn in AND one to each other, then the negation of the Predicate is computed in this way: it is simply the disjunction of multiple Predicates, each one representing the negation of a single element within each field.

Examples:
!{[10.0.0.1], [20.0.0.2], *, *} = {[!10.0.0.1], *, *, *} ∪ {*, [!20.0.0.2], *, *}
!{[10.0.0.*, !10.0.0.1], [20.0.0.2], *, [80], *} = {[!10.0.0.*], *, *, *, * } ∪ { [10.0.0.1], *, *, *, * } ∪ { *, *, *, [!80], *}

So, what the code does is simply take each element within each list, compute its negation (i.e., replace the value of the attribute neg with its complement) and then create a Predicate with that only field specified, and the others expressed with the wildcard. NOTE: for each scanned element, we have to first check if its value is different from the wildcard (!*equalStar*). In fact, the negation of a wildcard is the empty set. Any field in conjunction with the empty set returns the empty set itself, so these cases are not to be taken in consideration. NOTE2: for computing the negation of the protocol type list we call the method computeDifferenceL4ProtocolTypes that given the set of values corresponding to protoTypesList, returns its complement.

Chapter 6

Atomic Flows

This approach makes use of the concept of atomic predicates described in Chapter 2. The basic idea introduced by the authors, Yang and Lam, has been modified to suit our purposes. In particular, it is no longer used within the context of Verifying Network Reachability but to Verifying Satisfiability of a set of given NSRs. We use Atomic Predicates to represent the traffic that can cross the network and then, after the MaxSMT problem, to configure each firewall with rules expressed by certain atomic predicates.

According to the definition given by the two authors, Atomic Predicates are the smallest set of disjunct predicates such that each predicate, of the set over which they are computed, can be expressed as a disjunction of a subset of them.

The basic idea is to compute the set of atomic predicates representative of the network, based on what we will call "interesting" predicates, according to NSRs given in input by the user. We consider "interesting" all the predicates linked to nodes related to a requirement, such as the predicate representing the traffic generated by a node source of a requirement, the predicate representing the traffic that arrives in input to a destination node, but also predicates describing input traffic classes for transformers crossed on the path or describing conditions of rules on firewalls. Each one of these "interesting" predicates can be described as a disjunction of simpler and minimal atomic predicates, following the rules presented by the two authors and described in Chapter 2. Both atomic predicates and "interesting" predicates are modelled following the model presented in Chapter 5.

Then, after having computed the set B of atomic predicates, we proceed to generate for each requirement all related atomic flows, to give in input to the MaxSMT solver.

Definition 6.0.1 *A flow* $f = [n_s, t_{sa}, n_a, ..., n_h, t_{hi}, n_i, t_{ij}, n_j, ..., n_k, t_{kd}, n_d]$ *is defined atomic if each traffic* $t_{ij} \in B$*, where* B *is the set of atomic predicates computed above.*

Because of atomic predicates are totally disjointed one from each other, we can identify each of them with an integer identifier, keeping all the advantages of working with integers instead of complex Java classes.

6.1 Approach

In this thesis, we propose two new algorithms, for computing respectively the atomic predicates on the base of NSRs given in input, and the related atomic flows.

In order to understand these algorithms, we have to first introduce some notations. Each requirement r is expressed as a pair (C, a), where C is a condition and a is the action that must be performed on the flows that satisfy C. Each condition C is a Predicate modelled as the class described in Chapter 5 and representing the IP quintuple = {IPSrc, IPDst, pSrc, pDst, tProto}. Each action a is one of the two elements {ALLOW, DENY}. The "." notation, when applied to a tuple, is used to retrieve a specific tuple field. A flow satisfies C if the three following properties are satisfied: 1) its source and destination endpoints have IP address matching respectively C.IPSrc and C.IPDst, 2) its source traffic satisfies C.IPSrc and C.pSrc (that means it matches with the predicate {C.IPSrc, *, C.pSrc, *, *}, 3) its destination traffic satisfies C.IPDst, C.pDst and C.tProto (that means it matches with the predicate {*, C.IPDst, *, C.pDst, C.tProto}).

First of all, for each requirement r, all the network paths with endpoints e_s and e_d (where e_s and e_d are the nodes satisfying property 1) are computed. Each network path is thus represented by the two endpoints and a list of middleboxes $L = [n_1, n_2, \ldots, n_m]$. We save in N_a the list of all the transformers that are crossed by at least one path.

Algorithm 3 for computing the atomic predicates

Input: a set of n requirements, a set of m transformers $\mathcal{N}_a$
Output: the set of atomic predicates $\mathcal{B}$

1: $\mathcal{P} \leftarrow$ {false}
2: **for** $i = 0,1,...,n$ **do**
3: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{r_i.C.\text{IPSrc}, *, r_i.C.\text{pSrc}, *, *)\}$
4: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{*, *, r_i.C.\text{IPDst}, r_i.C.\text{pDst}, r_i.C.\text{tProto}\}$
5: **end for**
6: **for** $i = 0,1,...,m$ **do**
7: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \{\mathcal{I}_i^a, \mathcal{I}_i^d\} \cup \{\mathcal{D}_{i1}, \mathcal{D}_{i2}, \mathcal{D}_{i3}\})$
8: **end for**
9: $\mathcal{B} \leftarrow \mathcal{A}(\mathcal{P})$
10: $\mathcal{R} \leftarrow$ {false}
11: **for** $i = 0,1,...,m$ **do**
12: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{\mathcal{T}_i(b)|$ for each $b \in \mathcal{B}\}$
13: **end for**
14: **if** $\mathcal{B} = \mathcal{A}(\mathcal{P} \cup \mathcal{R})$ **then**
15: $\quad$ **return** $\mathcal{B}$
16: **else**
17: $\quad \mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{R}$, $\mathcal{B} \leftarrow \mathcal{A}(\mathcal{P})$
18: $\quad$ **goto** line 9
19: **end if**

Algorithm 4 for computing the atomic flows

Input: one requirement r, the source and destination nodes e_s and e_d, a list of middleboxes $\mathcal{L}$=[n_1,n_2,...,n_m], the set $\mathcal{B}$ of atomic predicates computed in Algorithm 1
Output: a set of atomic flows F_a

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

10: **function** RecursiveGen(i, b)
11: **if** $i == m + 1$ **then**
12: **if** b.IPDst $== \alpha(e_d)$ **then**
13: **return** $\{[e_d]\}$
14: **else return** $\emptyset$
15: **end if**
16: **end if**
17: **if** b.IPDst $== \alpha(n_i)$ **then return** $\emptyset$
18: **end if**
19: $t \leftarrow \mathcal{T}_i(b)$
20: $\mathcal{B}_t \leftarrow \{b_1,b_2,...,b_{m_t}\}$ such that $\vee_{j=1}^{m_t} b_j = t$ and $b_j \in \mathcal{B}_i$
21: $F_t \leftarrow \emptyset$
22: **for** $b_t \in \mathcal{B}_t$ **do**
23: **for** $f \in$ RecursiveGen$(i + 1, b_t)$ **do**
24: $f_t \leftarrow [n_i, b_t] + f$
25: $F_t \leftarrow F_t \cup \{f_t\}$
26: **end for**
27: **end for**
28: **return** F_t
29: **end function**

We run Algorithm 3 for creating, at first, the set P, containing the "interesting" predicates of the network (up to line 7), then for transforming the computed set P into the corresponding set B of atomic predicates, unique for the entire network, applying function A described in Chapter 2.

For each requirement r, we generate the Predicate representing the source traffic (according to property 2) and the Predicate representing the destination traffic (according to property 3) and we insert them into P. Then, for each transformer belonging to N_a we compute its forwarding domain $\{I_a, I_d\}$ and its transformation behaviour $\{D_1, D_2, D_3\}$ according to the algorithms presented in Chapter 2. We

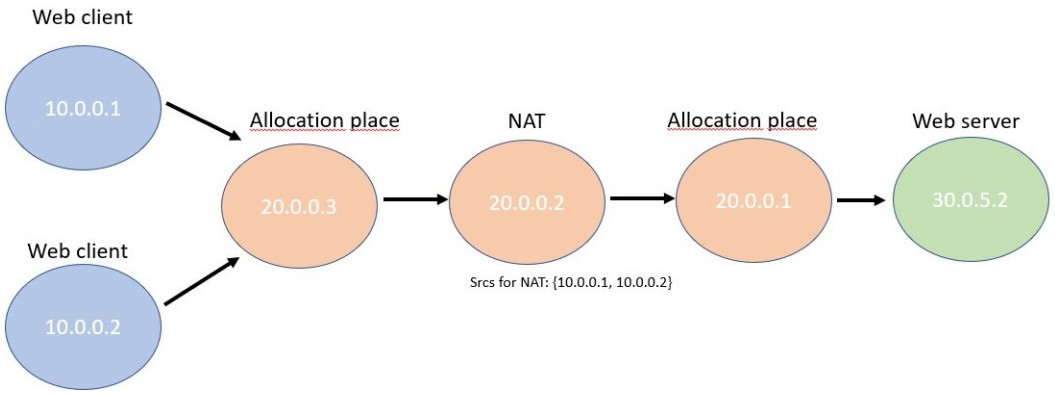
insert all these Predicates inside P. In line 7, we transform the set P into the corresponding set B of atomic predicates. Notice that we have not yet considered how predicates are transformed crossing the various transformers, P so far only contains the input packet classes for transformers but how predicates matching these classes are transformed has not yet been considered. So, starting from line 8, we must apply to each predicate inside B, that matches a specific input classes of a transformer included into N_a , the corresponding transformation T, obtaining in this way the transformed predicate. In other words, we insert into R the result of each predicate of B after having passed through a transform node. This is an iterative function that continues until we have in B a set of predicates representative for both input predicates and transformed ones. In line 12, if the condition $B = A(P' \cup R)$ is satisfied it means that each transformed predicate in R is already equal to the disjunction of a subset of predicates contained in B, so we can stop the iterative algorithm, since we have in B the set representative for input predicates and transformed ones.

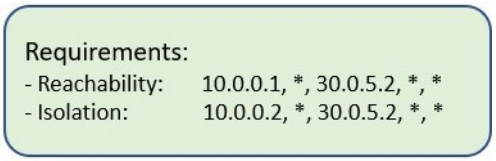
The result of Algorithm 3, that is the set of atomic predicates representative for the networks and for the requirements, is then used as input for Algorithm 4.

We pass to consider one requirement and one path at a time. The traffic generated by e_s, source node of the requirement, is grouped in a subset B_0, that represents the disjunction of all predicates of B whose IPSrc and pSrc are equal to IPSrc and pSrc expressed by the condition of the requirement. Starting from B_0, we compute recursively all related atomic flows. Each single atomic predicate b, belonging to 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 disjointed atomic predicates. At each recursion level, some pruning is done in order to reduce the total number of atomic flows, discarding the ones that are not part of the solution, because they are incorrect (i.e., they do not arrive at destination with the correct IPDst, pDst and tProto that matches the condition of the requirement) or because they are dropped during the path.

6.2 Example

Let us consider a clarifying example. In order to compute Atomic Predicates, we use formulas described in Chapter 2.





Each requirement follows only one path:

- path for requirement 1: [10.0.0.1, 20.0.0.3, 20.0.0.2, 20.0.0.1, 30.0.5.2]
- path for requirement 2: [10.0.0.2, 20.0.0.3, 20.0.0.2, 20.0.0.1, 30.0.5.2]

Set N_a containing transformer: {20.0.0.2}

Line 1-4 of Algorithm 3: compute predicates representing source and destination traffic, each of these predicates are inserted into P

- $S_1 = \{10.0.0.1, *, *, *, *\}$ from source of requirement 1
- $S_2 = \{10.0.0.2, *, *, *, *\}$ from source of requirement 2
- $D_1 = \{*, *, 30.0.5.2, *, *\}$ from destination of requirements 1 and 2

Line 5-6 of Algorithm 1: compute the forwarding behaviour for the nodes and input classes for transformers. In this example, there are no nodes (i.e., firewall) dropping packets, so all packets are simply forwarded by all nodes, and there is only a transformer that is the NAT in 20.0.0.2. Let us consider its input classes and then add them to P

- $D_1 = \{10.0.0.1, *, !10.0.0.1 \wedge !10.0.0.2, *, *\} \cup \{10.0.0.2, *, !10.0.0.1 \wedge !10.0.0.2, *, *\}$
- $D_2 = \{!10.0.0.1 \wedge !10.0.0.2, *, 20.0.0.2, *, *\}$
- $D_3 = \neg D_2 \wedge \neg D_3$

D_1 refers to input classes for shadowing operation: each packet arriving to the NAT matching one of these two predicates is shadowed (that means, its source IP address is translated into the IP address of the NAT, 20.0.0.2). D_2 refers to the operation of reconversion. Predicates matching D_3 instead are simply forwarded.

Line 7: add these three predicates inside P and then compute $B = A(P)$.

$P = \{10.0.0.1, *, *, *, *\} \cup \{10.0.0.2, *, *, *, *\} \cup \{*, *, 30.0.5.2, *, *\} \cup \{10.0.0.1, *, !10.0.0.1 \wedge !10.0.0.2, *, *\} \cup \{10.0.0.2, *, !10.0.0.1 \wedge !10.0.0.2, *, *\} \cup \{!10.0.0.1 \wedge !10.0.0.2, *, 20.0.0.2, *, *\}$.

$B = A(P) = \{10.0.0.1, *, 10.0.0.1, *, *\} \cup \{10.0.0.1, *, 10.0.0.2, *, *\} \cup \{10.0.0.1, *, 20.0.0.2, *, *\} \cup \{10.0.0.1, *, 30.0.5.2, *, *\} \cup \{10.0.0.1, *, !10.0.0.1 \wedge !10.0.0.2 \wedge !20.0.0.2 \wedge !30.0.5.2, *, *\} \cup$
$\{10.0.0.2, *, 10.0.0.1, *, *\} \cup \{10.0.0.2, *, 10.0.0.2, *, *\} \cup \{10.0.0.2, *, 20.0.0.2, *, *\} \cup \{10.0.0.2, *, 30.0.5.2, *, *\} \cup \{10.0.0.2, *, !10.0.0.1 \wedge !10.0.0.2 \wedge !20.0.0.2 \wedge$

!30.0.5.2, *, *} ∪
{!10.0.0.1 ∧ !10.0.0.2, *, 10.0.0.1, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2, *, 10.0.0.2, *, *} ∪
{!10.0.0.1 ∧ !10.0.0.2, *, 20.0.0.2, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2, *, 30.0.5.2, *, *
} ∪ {!10.0.0.1 ∧ !10.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}

These are the atomic predicates representing sources and destinations of the requirements and the input classes for transformers. Now we must compute how these predicates are transformed crossing the NAT. Each predicate is put in intersection with D_1, D_2 and D_3, if the intersection with one of these classes exists, then the corresponding transformation is applied.

In particular,
{10.0.0.1, *, 30.0.5.2, *, *}, {10.0.0.1, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}, {10.0.0.2, *, 30.0.5.2, *, *}, {10.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *} matches with D1.

The corresponding transformed predicates after applying Shadowing are {20.0.0.2, *, 30.0.5.2, *,*} and {20.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}.

{!10.0.0.1 ∧ !10.0.0.2, *, 20.0.0.2, *, *} instead matches with D_2.

The corresponding transformed predicates after applying Reconversion are {!10.0.0.1 ∧ !10.0.0.2, *, 10.0.0.1, *, *}, {!10.0.0.1 ∧ !10.0.0.2, *, 10.0.0.2, *, *}.

We add the resulting transformed predicates to R and then we compute $B = A(P \cup R)$.

R = {20.0.0.2, *, 30.0.5.2, *,*} ∪ {20.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2, *, 10.0.0.1, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2, *, 10.0.0.2, *, *}.

$B = A(P \cup R)$ = {10.0.0.1, *, 10.0.0.1, *, *} ∪ {10.0.0.1, *, 10.0.0.2, *, *} ∪ {10.0.0.1, *, 20.0.0.2, *, *} ∪ {10.0.0.1, *, 30.0.5.2, *, *} ∪ {10.0.0.1, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *} ∪ {10.0.0.2, *, 10.0.0.1, *, *} ∪ {10.0.0.2, *, 10.0.0.2, *, *}∪{10.0.0.2, *, 20.0.0.2, *, *}∪{10.0.0.2, *, 30.0.5.2, *, *}∪{10.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *} ∪ {20.0.0.2, *, 10.0.0.1, *, *}∪{20.0.0.2, *, 10.0.0.2, *, *}∪{20.0.0.2, *, 20.0.0.2, *, *}∪{20.0.0.2, *, 30.0.5.2, *, *}∪{20.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 10.0.0.1, *, *}∪{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 10.0.0.2, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 20.0.0.2, *, *} ∪ {!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 30.0.5.2, *, *}∪{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}.

These are the final atomic predicates representative of the network. As we can see they are simple, minimal, and disjunct so we can identify them with an integer identifier.
{10.0.0.1, *, 10.0.0.1, *, *}(1)
{10.0.0.1, *, 10.0.0.2, *, *}(2)
{10.0.0.1, *, 20.0.0.2, *, *}(3)
{10.0.0.1, *, 30.0.5.2, *, *}(4)
{10.0.0.1, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}(5)
{10.0.0.2, *, 10.0.0.1, *, *}(6)
{10.0.0.2, *, 10.0.0.2, *, *}(7)

{10.0.0.2, *, 20.0.0.2, *, *}(8)
{10.0.0.2, *, 30.0.5.2, *, *}(9)
{10.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}(10)
{20.0.0.2, *, 10.0.0.1, *, *}(11)
{20.0.0.2, *, 10.0.0.2, *, *}(12)
{20.0.0.2, *, 20.0.0.2, *, *}(13)
{20.0.0.2, *, 30.0.5.2, *, *}(14)
{20.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2 ∧ !30.0.5.2, *, *}(15)
{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 10.0.0.1, *, *}(16)
{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 10.0.0.2, *, *}(17)
{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 20.0.0.2, *, *}(18)
{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, 30.0.5.2, *, *}(19)
{!10.0.0.1 ∧ !10.0.0.2 ∧ !20.0.0.2, *, !10.0.0.1 ∧ !10.0.0.2 AND !20.0.0.2 ∧ !30.0.5.2, *, *}(20)

The transformation behaviour of the NAT can be modelled as follow:
(4) → (14) (Shadowing transformation)
(9) → (14) (Shadowing transformation)
(18) → (16) + (17) (Reconversion operation)
(3) → X
(8) → X
(13) → X

All the other atomic predicates are simply forwarded, without being transformed.
NOTE: (3), (8), (13), and (18) are not forwarded because the NAT (IP address 20.0.0.2) represents their destination (their IPDst is equal to 20.0.0.2 and they match only with D_3).
NOTE$_2$: In case a predicate arrives to the NAT matching with D_2, the reconversion is applied. In this case, the transformation returns two predicates, (16) and (17). We cannot know a priori which is the correct one, it depends on the destination of the path. So, the entering atomic flows is splitted into two different atomic flows and recursion continues, in one case with atomic predicates (16) exiting from the Nat, in the other with (17).

At this point we can run Algorithm 4, considering one requirement and one path at a time, to compute the set of atomic flows (list of alternating nodes and traffics).

Requirement 1: {10.0.0.1, *, 30.0.5.2, *, *}
Source traffic B_0 can be represented by atomic predicates (1), (2), (3), (4) and (5), which are the predicates whose IPSrc is equal to 10.0.0.1. Destination traffic instead is represented by (4), (9), (14) and (19), which are the predicates whose IPDst is equal to 30.0.5.2.

Starting from B_0, we propagate each atomic predicates along the path taking in consideration the forwarding and transformation behaviour of the nodes.

- [10.0.0.1, (1),, reached its destination without reaching the destination of the requirement, so this atomic flow must be discarded

- [10.0.0.1, (2), 20.0.0.3, (2), 20.0.0.2, (2), 20.0.0.1, (2), discarded at the destination because it arrives with IPDst different from the IP address of the destination (30.0.5.1).

- [10.0.0.1, (3), 20.0.0.3, (3), discarded because it reached its destination without reaching the destination of the requirement.

- **[10.0.0.1, (4), 20.0.0.3, (4), 20.0.0.2, (14), 20.0.0.1, (14), 30.0.5.2].** Accepted atomic flow. Notice how atomic predicate (4) is transformed crossing the NAT following the Shadowing operation.

- [10.0.0.1, (5), 20.0.0.3, (5), 20.0.0.2, (5), 20.0.0.1, (5)..., discarded at the destination

6.3 Advantages

1. Predicates are totally disjointed one from each other. The main benefit of this approach is to arrive to a definition of a set of predicates that are totally disjointed one from each other. If we configure a firewall with a deny rule whose condition is expressed by a certain atomic predicate, we can be sure that this rule will block only the traffic specified by that predicate, and all the other traffics will be allowed to pass, because they are totally disjointed from the blocked one.

2. Each atomic predicate can be identified with an integer, as suggested in Chapter 2, keeping all the advantages of working with integers instead of complex Java classes representing the IP header fields.

3. The MaxSMT formulation will then use integers for representing a traffic. One of the most critical aspects working with MaxSMT solver, z3 in our case, is understanding how to interface these tools with the classes to give them in input. In our case, we have complex Java classes, represented by the Predicate class. Z3 works mostly with simple data classes (integers, Booleans, characters, strings etc.), so it is often difficult to model complex Java classes using only the simple data types provided by the tool. With this solution, the MaxSMT formulation will simply use integers for representing a traffic, instead of taking in input multiple variables (e.g., four integers representing the bytes of the source IP address, other four for the destination IP address, etc.). This is expected to reduce the time required by z3 for solving the MaxSMT problem.

4. z3 will not see any details about the network. From its point of view, it is simply solving a problem on integers.

5. Algorithm 4, the one describing how atomic flows are computed starting from the set of atomic predicates, can be easily parallelized.

6.4 Disadvantages

1. Initial time spent for computing atomic predicates.
2. The output of the MaxSMT problem will be a list of disjointed configured rules, we cannot demand to z3 to merge more rules into a single one, because it is working with integers. This is not a problem if we adopt solutions such as firewall with hash-based rules, but on the other cases we must demand the work to a post-processing Java algorithm, that takes the rules configured on a firewall and merge them.

6.5 Other considerations

Why considering a unique set of atomic predicates related to all requirements (*approach 1*), and not n set of atomic predicates each one related to a single requirement (*approach 2*)?

- With *approach 1*, the complexity is $O(n + m)$ where n is the number of requirements while m the number of transformers. With *approach 2* the complexity is $O(n * m)$. Furthermore, most iterations will repeat the same operations (e.g., the computation of input packet classes D1, D2, D3 for a transformer in common with more requirements).
- Even if the total number of atomic predicates computed with *approach 2* is smaller than the number computed with *approach 1*, the atomic predicates could not be disjointed (for sure they are disjointed locally in the set of the single requirement, but not for sure for the resulting set given by merging them)

Chapter 7

Tests on Atomic Flows

The goal of this section is to introduce the various test cases used to evaluate the performance of the two approaches. All test cases described in this Chapter are the same that will be used later to evaluate the performance of the other approach, based on Maximal Flows and described in Chapter 8. In particular, we will evaluate how much time the two approaches take to compute the set of traffic flows and how the number of generated flows varies increasing the size of the network and the number of transformers inside it. We still do NOT take into account the time taken to solve the MaxSMT problem. A total analysis, including the time taken by the solver, will be done later in Chapter 10.

7.1 Test parameters

Parameters that can be configured before running the test are number of requirements (**REQ**), number of web clients (**WC**), number of web servers (**WS**), number of NATs (**NATs**), number of firewalls (**FWs**), number of sources present in each NAT (**NATSrcs**), number of rules present in each firewall (**FWRules**) and **percentage** of requirements with information also on ports and prototype (instead of simply having information about source and destination of the requirement). It is also possible to specify through a flag (**true/false**) whether the rules for pre-existing firewalls are to be build automatically starting from source/destination of the requirements (in this case we will have rules on firewalls only affecting nodes on which at least one requirement has been built) or randomly selecting any nodes from the network (in this case not necessarily belonging to a requirement). Based on this last decision, time can vary greatly. In fact, rules that have source/destination not belonging to any requirement introduce new "interesting" predicates on the network. At the contrary, if source/destination of the rules are created by looking at the requirements, no additional predicates are created (they have all already been added to the set when considering source/destination of the requirements).

As for times considered in analysing the approach based on Atomic Flows, they differ in: **time to generate the atomic predicates**, time to associate each transformation behaviour to the corresponding transformer (each transformer in fact, as we have seen in the example, has an internal map the specify haw

that node has to behave when a certain atomic predicates arrives to it in input: example, atomic predicate X arrives in input and it must be transformed into atomic predicate Y, the time taken to fill these maps is not irrelevant) and **time to generate the atomic flows**. The **number of atomic predicates generated** for each test is also analysed.

7.2 Tests execution

We basically followed two approaches.

The first consisted of finding a starting value for each configurable parameter. Once this configuration has been obtained, let us call it "base configuration", we proceed to execute the various tests increasing one parameter at a time, keeping all the others at their basic value. This approach was mainly used to understand which parameters have the greatest influence on the number of generated atomic predicates, and consequently on the total execution time. We set the standard configuration to the case with "**100 requirements, 100 web clients, 100 web servers, 25 NAT, 25 FW, 10 sources for each NAT, 10 rules for each firewall**" (case which takes 4.7 seconds to complete in case fw rules are not taken from requirement, so in the worst case). NOTE: for the moment we do not consider the aspect related to the percentage of requirements with indication on ports and prototypes, which deserves a separated discussion, and we will talk about it at the end of this Chapter.

The second approach, instead, consisted in the progressive increase of all parameters at the same time. So, a sort of progressive enlargement of the network, in which the ratios between the values of the parameters is maintained constant, while they are increased together step by step.

7.3 Analysis of test results

Most affecting parameters

As we expected, we found that parameters most affecting the number of generated atomic predicates, and consequently affecting the total execution time, are the **number of requirements**, the **number of NATs**, the **number of firewalls** (when the rules are NOT generated starting from requirements, i.e., **FALSE configuration**) and the **number of rules within those firewalls**.

Number of requirements because it is from the requirements that we start to build the atomic predicates: a predicate is generated for the source traffic of each requirement and a predicate for the destination traffic of each requirement. The more the requirements are, the more the starting "interesting" predicates, and consequently the more time is required to transform them into atomic predicates. Moreover, the number of generated atomic flows will be certainly greater, increasing in this way also the time to generate them. We must generate all flows for each

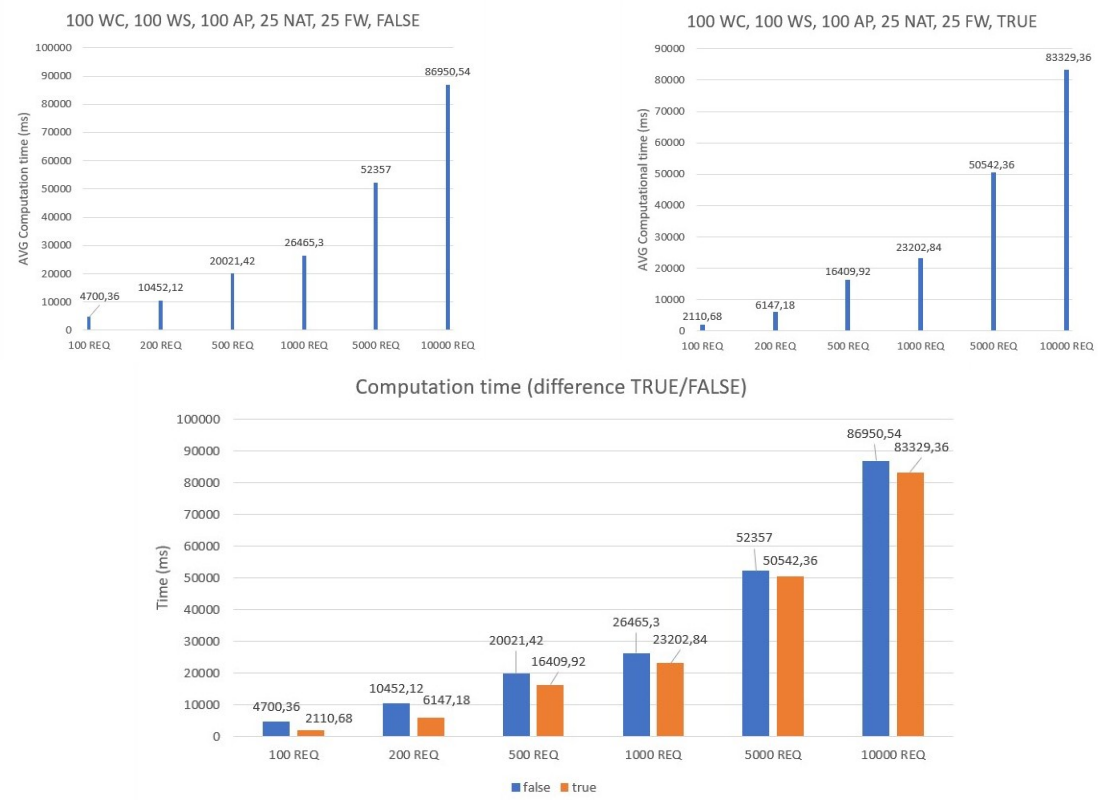


Figure 7.1: Computation time VS number of requirements

specified requirement, so the higher the number of requirements, surely the higher the number of atomic flows that will be generated.

We can see, from the third graphic of Figure 7.1, that the difference in time, between the case in which fw rules are generated from requirements ("TRUE case") and the case in which they do not ("FALSE case"), is really minimal. It will be much more decisive for other parameters. This is reasonable since the number of considered requirements is very high, and so there is a high probability that, even in the "FALSE case", fw rules coincide with the source/destination of a requirement.

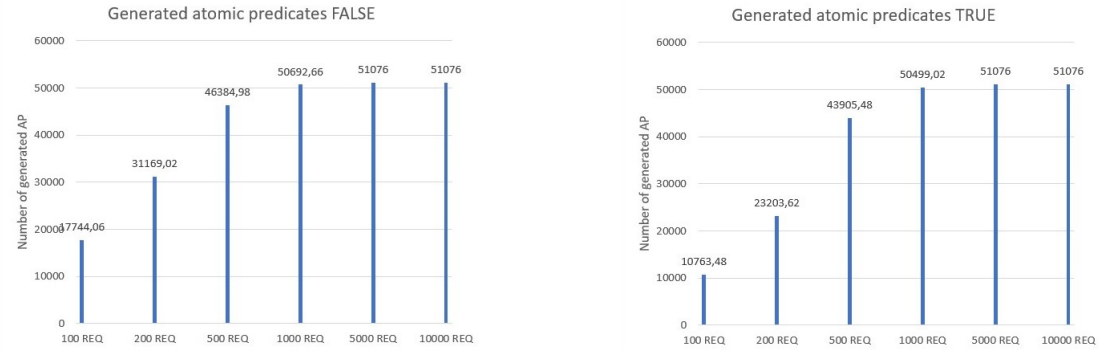


Figure 7.2: Number of generated AP VS number of requirements

We can also see, with these last two graphs (Figure 7.2), that, with a high number of requirements, the number of generated requirements saturates to the maximum possible value (51076). This happens when all the nodes of the network

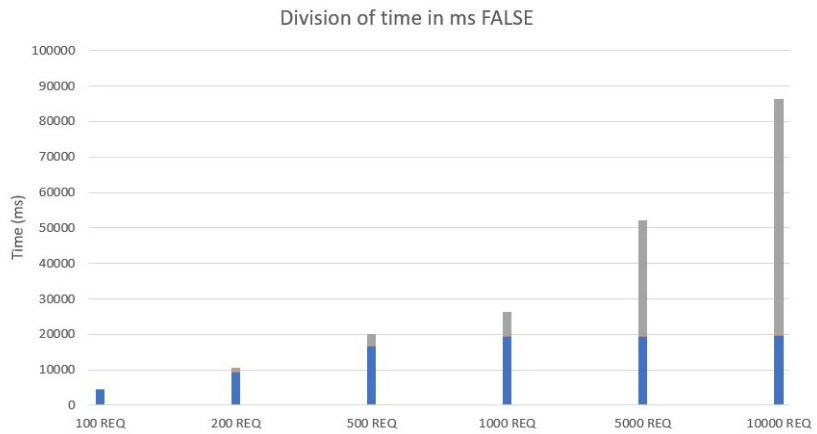


Figure 7.3: Division of time VS number of requirements

are related to at least one requirement and therefore used for the computation of the atomic predicates. In this way, the set representing the whole network is computed, no node excluded. As we can see from the last graph (Figure 7.3), for the last three columns the time taken to compute the set of atomic predicates (coloured BLUE) remain constant and equals to the maximum possible value, and only the time used to compute the set of related atomic flows (coloured GREY) increases, because of the major number of requirements.

Number of NATs because they introduce transformations. So, in addition to predicates representing source and destination of requirements, it must also be considered how they are transformed, and which are the input classes for the NAT. Number of generated atomic predicates increases. All three considered times increase: both the time to generate atomic predicate (because the initial set of "interesting" predicates is bigger), the time to build maps for the transformers (because there are an higher number of transformers) and the time to compute atomic flows (because there are more atomic predicates and therefore the recursive function is called more times).

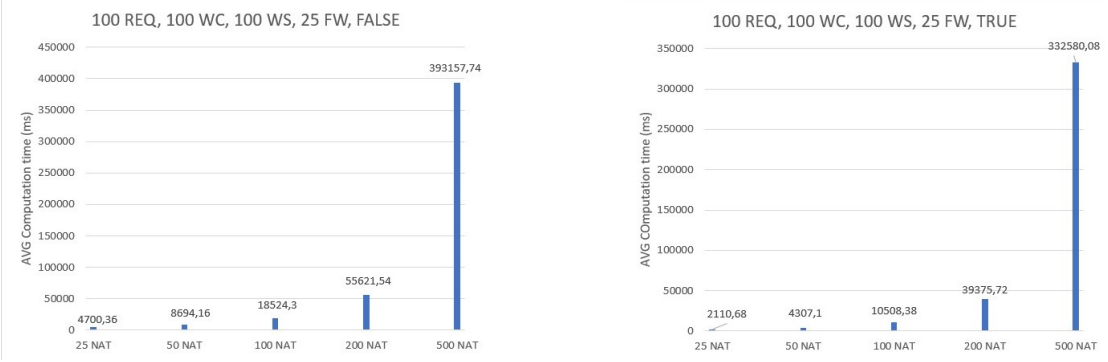


Figure 7.4: Computation time VS number of NATs (i)

Here too, as we can see in Figure 7.5, the difference in time between the true and false case is not so marked, since the number of firewall present in the network (25) is not so decisive and does not determine such a substantial increase in the number of APs obtained from their rules.

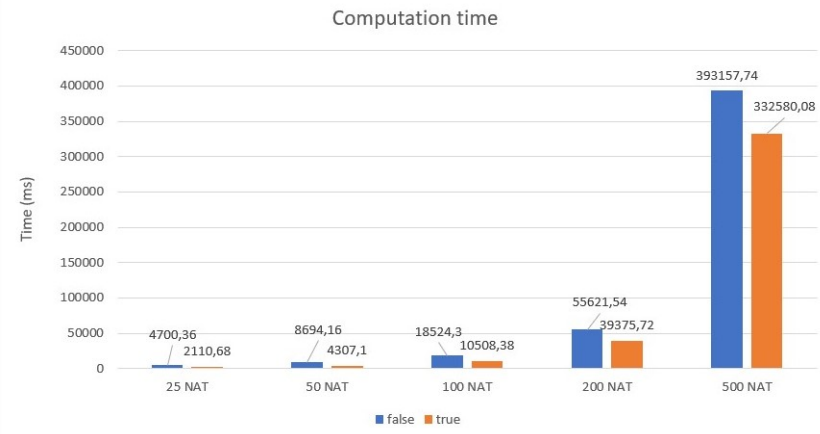


Figure 7.5: Computation time VS number of NATs (ii)

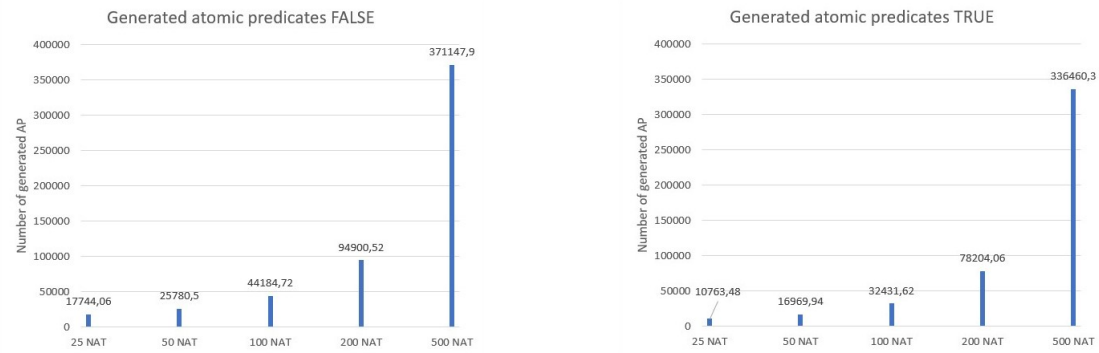


Figure 7.6: Number of generated AP VS number of requirements

As we can see in Figure 7.6, the number of NATs is decisive in increasing the number of generated atomic predicates. Among those analysed, it is the parameter the most influences the increase because it is also the only transformer considered.

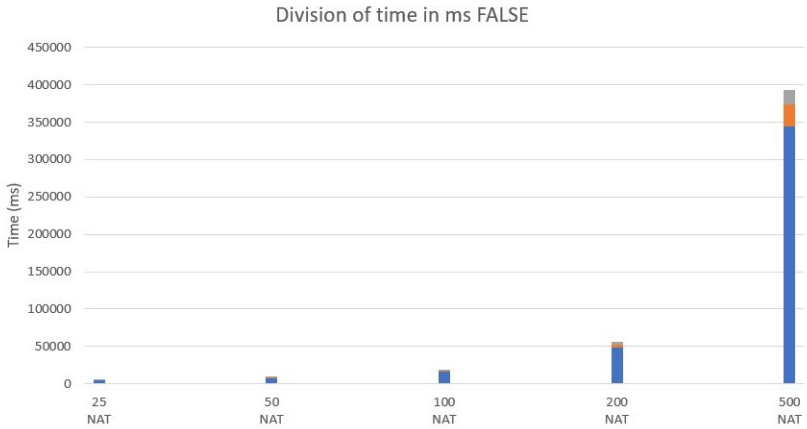


Figure 7.7: Division of time VS number of NATs

Number of FWs in case configuration is FALSE because they increase the number of generated atomic predicates for the network. For each firewall we must build the Allowed and Denied set of predicates Ia, Id, characterizing the forwarding

behaviour of the node. Each predicate present in these two sets Ia, Id must then be added to the set of "interesting" predicates. In case configuration would be TRUE, these predicates are already present in the set (added because they are source/destination of a requirement) and so should not be added again. But if the configuration is FALSE, the corresponding new "interesting" predicates must be added because they are not yet present determining an increase of generated atomic predicates. So, the more FWs in a false configuration and the **more rules configured on them** imply longer times.

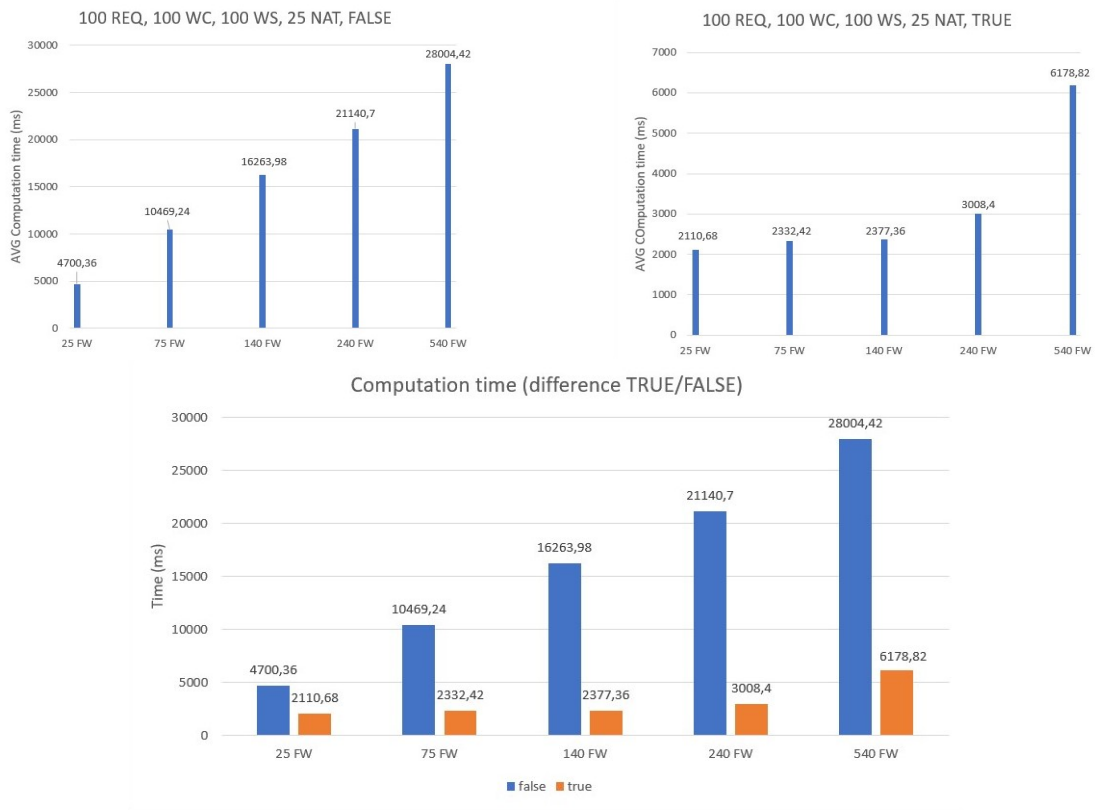


Figure 7.8: Computation time VS number of Firewalls

NOTE : the unusual numbers (25, 75, 140, 240, 540) represent the total number of firewalls in the network. Only a part of them is actually crossed by at least a flow. For how the network is automatically built, these numbers are those necessary so that the average number of firewall crossed is equals to 25, 50, 100, 200, 500 (i.e, same number used for progression tests on NATs).

In case of configuration equal to FALSE, the increase in time is decisive. If rules on firewalls do not concern nodes included yet in the requirements, the computation will be much longer. NOTE: the increase in times in case configuration is equal to TRUE only concerns the time spent to process the rules for the construction of the Allowed and Denied list. Then, in the "TRUE case", the predicates in these list are not inserted in the list of "interesting" predicates because they are already present and so time does not increase further.

As we can see in Figure 7.9, in the "True case", the number of generated atomic predicates does not increase but remains almost constant (even if the number of firewalls increases).

Figure 7.9: Number of generated AP VS number of Firewalls

Number of rules configured on firewalls with FALSE configuration. As mentioned before, if firewalls have rules taken selecting nodes randomly (and not from requirement), then the number of atomic predicates increases.

As we can see, from the following graphs, in the TRUE case, the number of atomic predicates remains exactly the same. Times are slightly different because of the time it takes to process and create the Allowed and Denied list {Ia, Id}, since we have more rules inserted into firewalls.

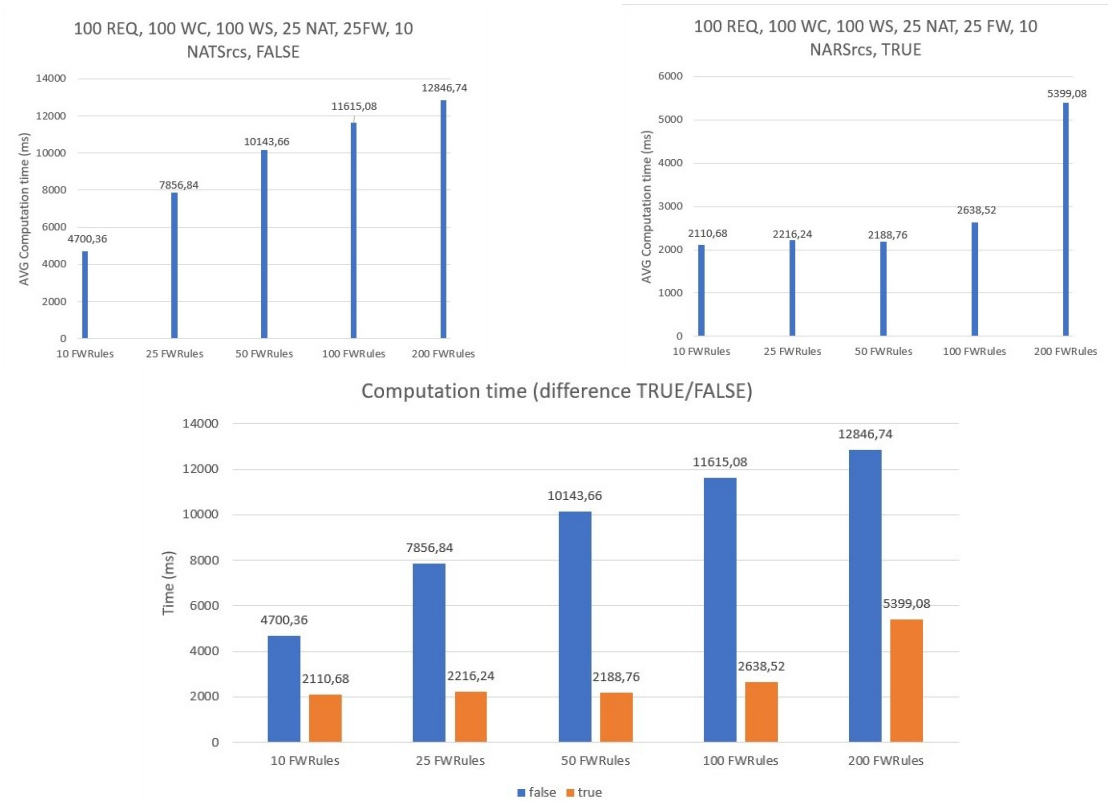


Figure 7.10: Computation time VS number of Firewall rules

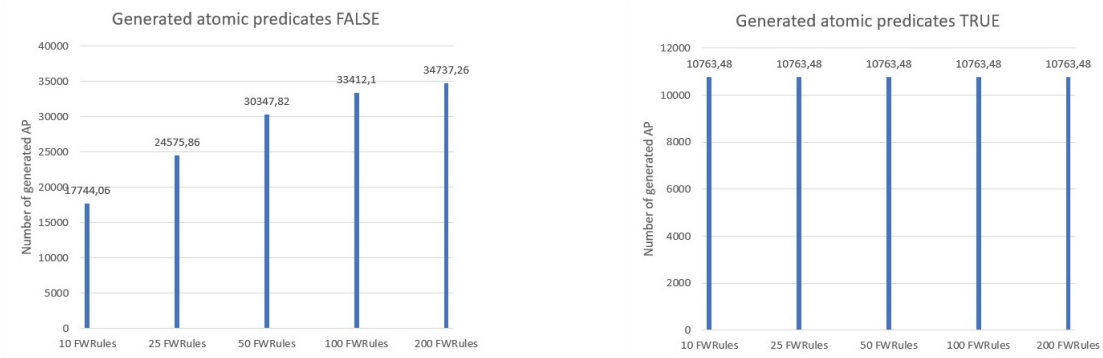


Figure 7.11: Number of generated AP VS number of Firewall rules

Parameters that do not affect or have minimal influence

Among the parameters that do not greatly affect times there are **number of web clients and web servers**, **number of firewalls with configuration equal to TRUE** and **number of NAT sources**.

Number of Web Clients and Web Servers. Like Allocation Places, they do not affect times because the number of atomic predicates depends on the number of requirements and not on the number of nodes in the network. This means that an "interesting" predicate is not created for each node of the network (as it is done in the Yang and Lam Algorithm), but only on the basis of requirements (said in another way on the basis of source and destination nodes of requirements). ANOMALY: from the figures we can see a sort of anomaly. Increasing the number of web clients and web servers seems that the total time decreases. This actually cannot be true, and the cause can be found in the rough creation of the network. In fact, looking at the data of the number of crossed firewalls, it seems that increasing the number of clients and servers determines a decreasing of the number of firewalls crossed.

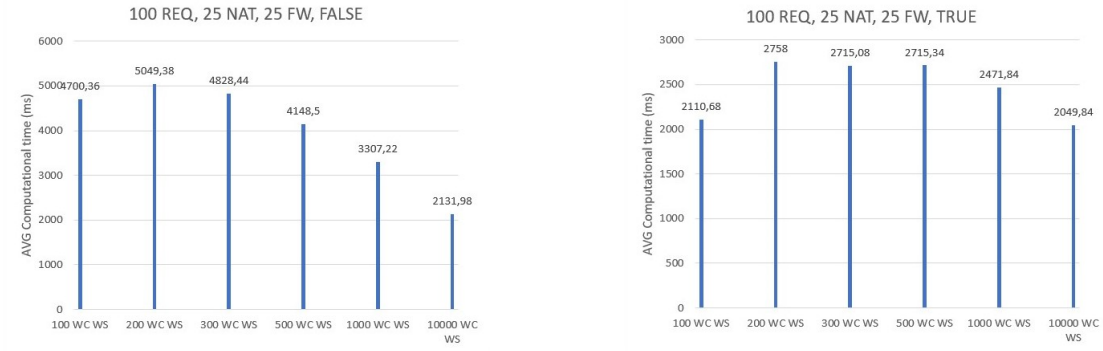


Figure 7.12: Computation time AP VS number of web clients and web servers

The difference is especially noticeable in the "FALSE case" and it is reasonable. In fact, that is the case in which firewalls and their rules, after being processed, increase the number of APs. In the "TRUE case" instead, a greater number of firewalls crossed simply implies a greater processing time for processing the Allowed and Denied list (so a less significant increase because the number of APs remains constant).

Number of NAT sources for each NAT. They do not increase the number of generated APs (it remains exactly the same). Remember that an atomic predicate computed on predicate X, also automatically generates the negation of X. So, if we have a set of 100 addresses, the number of APs remains the same, both in the case 98 are obscured by the Shadowing operation and in the case only 20 are obscured, because the algorithm also generates the transformations for the denied set (for the 2 and 80 addresses respectively that are not included between NAT sources). The real discriminant is the fact that NAT exists or not (and therefore it is the number of NATs to influence), which must implies whether the transformation must be introduced or not.

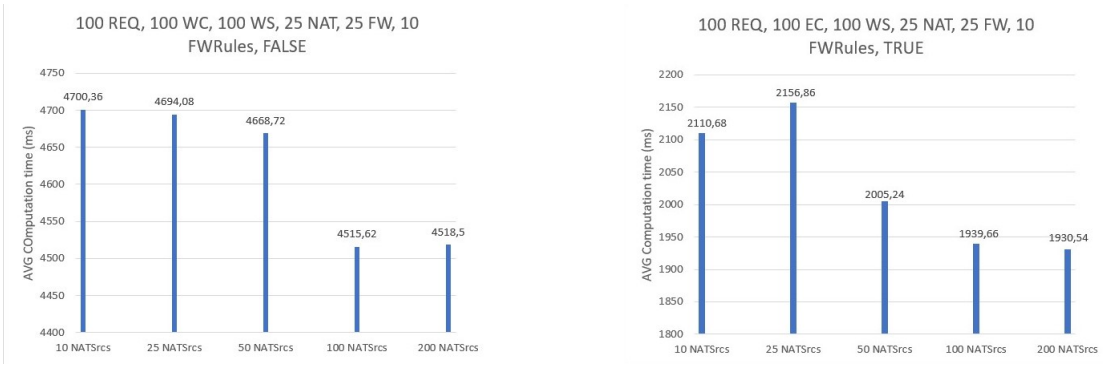


Figure 7.13: Computation time VS number of NAT sources

We can see that times remain almost constant, both in the TRUE case and in the FALSE case. They decrease slightly, but one again the causes are to be found in how the network is built.

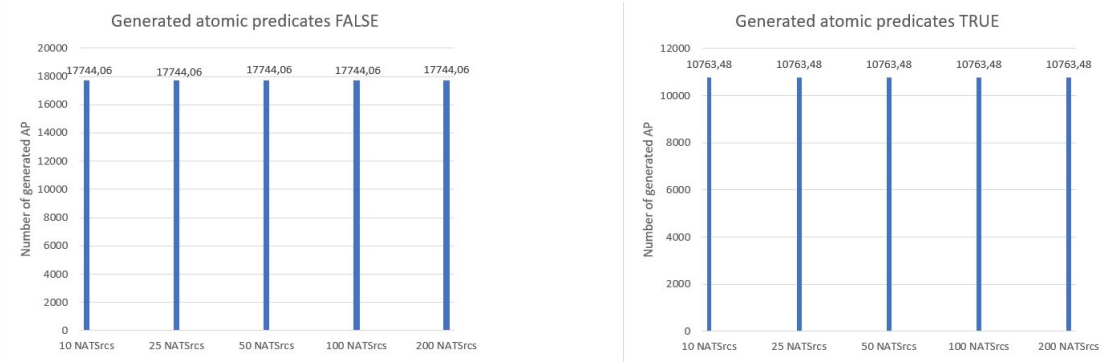


Figure 7.14: Number of generated AP VS number of NAT sources

The number of generated atomic predicates instead remains exactly the same in both cases.

Number of firewalls with TRUE configuration and their **number of rules**. See above.

Progression tests

Figure 7.15: Computation time VS progression tests

Other tests

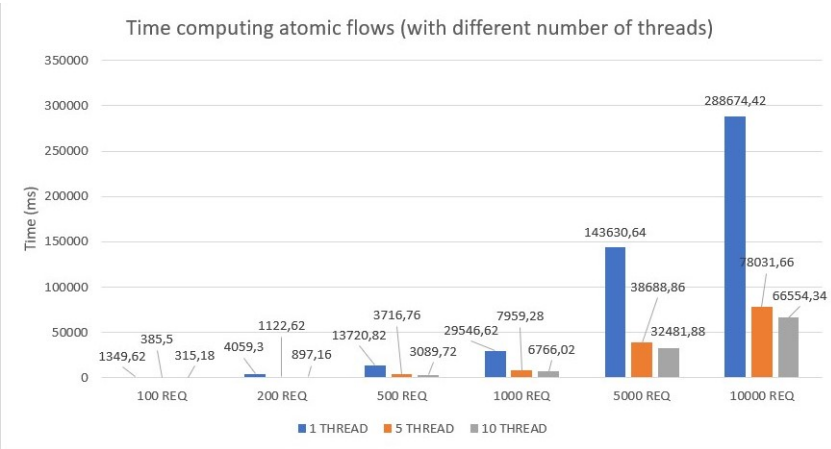


Figure 7.16: Computation time VS number of threads

This graph shows some statistics obtained exploiting parallelization for the atomic flows computation phase.

Percentage of requirements with information on ports and prototype

We found that an increase in the number of requirements that also contain information on port numbers and L4 prototype has drastic influence on the total execution time, because it drastically increases the number of generated APs. Let us consider the base case which does not contemplate the information about ports and prototype. It generates on average 11108 predicates, expressed by the quintuple {IPSrc, *, IPDst, *, *}. If we introduce even just one port number Z for the source of a requirement, this would be enough to double the number of generated atomic predicates: they would be 11108 with source port equal to Z and 11108 with source port !Z. This leads to an exponential increase in the number of generated atomic predicates and therefore in the total time.

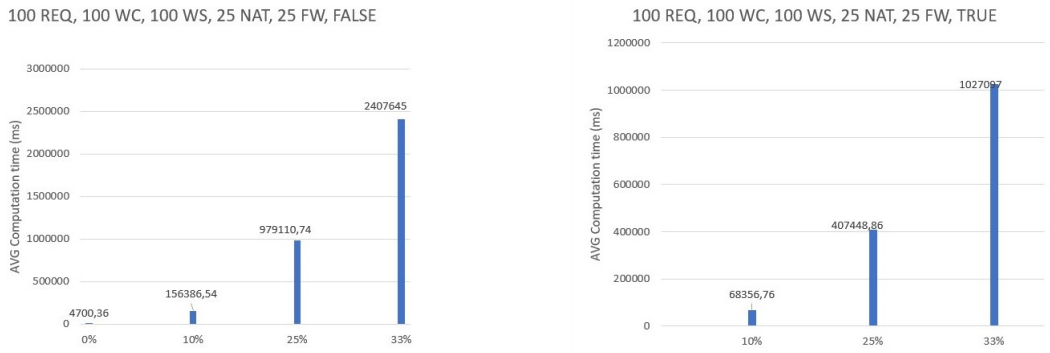


Figure 7.17: Computation time VS percentage of requirements with information on ports and protocol type

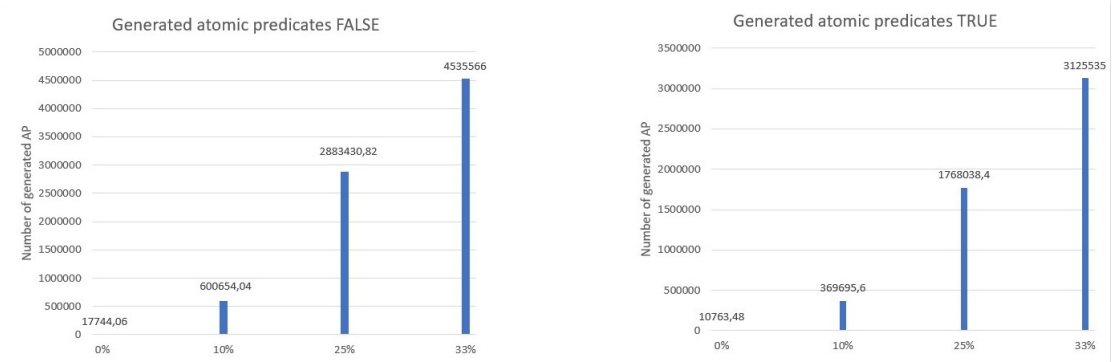


Figure 7.18: Number of generated AP VS percentage of requirements with information on ports and protocol type

The problem is the fact that, if we have a requirement {IPSrc X, Z, IPDst Y, *, *}, it is not possible to associate source port Z only to predicates having source address = X and destination address = Y. This information on the port number is propagated to all other predicates having source and destination address different from X and Y. This is how the model proposed by Yang and Lam is built. A solution could be not considering the port numbers in the atomic predicate if, on the paths of the requirements, there are not transformers working on port numbers. If the transformer works only on IPAddresses, that we cannot consider the information on port numbers for the computation of the set of atomic predicates.

Chapter 8

Maximal Flows

The second proposed approach consists of **Maximal Flows**. If with Atomic Flows we tried to split as much as possible each traffic flow into minimal and disjoint traffics, that can be subsequently identified with an integer identifier, with this approach we try to do the opposite. That is, 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.

definition

Definition 8.0.1 *Maximal Flows. 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. All the flows of F_r that are not in F_r^M are subflows of flows that are in F_r^M. This means we are trying to aggregate as much as possible different flows into maximal flows representative for all the ones that have been joined. All flows represented by the same maximal flow behave in the same way when crossing the various nodes of the network, so that it is sufficient to consider the maximal flow and not each single flow that it represents.

Also in this case, flows are modeled as a list of alternating nodes and Predicates, representing the traffic traveling between two consecutive nodes. Predicates contained inside a Maximal Flow are no longer necessary Atomic but express the disjunction of several quintuples.

8.1 Approach

Before formulating the MaxSMT problem, F_r^M is computed for each NSR r on the basis of the transformation behaviour of network functions, by means of Algorithm 5.

Given the allocation graph, G_A, the set $paths(r, G_A)$, containing the paths of G_A that satisfy $r.C$ is computed (line 2). Each path is represented as a list of nodes

Algorithm 5 computation of F_r^M

Input: a requirement r, and an AG G_A
Output: F_r^M

1: $F_r^M = \emptyset$
2: **for each** $p = [n_0, n_1, ..., n_{m+1}] \in paths(r, G_A)$ **do**
3: $F \leftarrow \{[n_0, t_1^r, n_1, \text{true}, n_2, ..., \text{true}, n_{m+1}]\}$
4: **for** $i = 1,2, ..., m$ **do**
5: $F \leftarrow \{l + [b_i \wedge b_i', n_i] + l' \mid l + [b_i, n_i] + l' \in F,$
6: $b_i' \in \{\mathcal{I}_i^a, \mathcal{I}_i^d\}\}$
7: $F \leftarrow \{l + [b_i \wedge b_i', n_i] + l' \mid l + [b_i, n_i] + l' \in F,$
8: $b_i' \in \{\mathcal{D}_{ij}\}\}$
9: $F \leftarrow \{l + [b_i, n_i, b_{i+1} \wedge \mathcal{T}_i(b_i), n_{i+1}] + l' \mid$
10: $l + [b_i, n_i, b_{i+1}, n_{i+1}] + l' \in F\}$
11: **end for**
12: $F' \leftarrow \{l + [t_{m+1}^r \wedge b_{m+1}, n_{m+1}] \mid l + [b_{m+1}, n_{m+1}] \in F\}$
13: **for** $i = m, m-1, ...,1$ **do**
14: $F' \leftarrow \{l + [b_i \wedge \mathcal{T}_i^{-1}(b_{i+1}), n_i, b_{i+1}] + l' \mid$
15: $l + [b_i, n_i, b_{i+1}] + l' \in F'\}$
16: **end for**
17: **if** $F \neq F'$ **then**
18: $F \leftarrow F'$
19: **goto** line 4
20: **end if**
21: $F_r^M \leftarrow F_r^M \cup F$
22: **end for**
23: **return** F_r^M

$p = [n_0, n_1, n \ldots]$ in which the endpoints, e_S and e_d, have the IPAddress equals respectively to the IPSrc and IPDst expressed in the condition of the requirement.

For each path, all the corresponding Maximal Flows are computed and added to the result set. This computation is performed iteratively. At each iteration, two set of lists F and F' are computed (these are lists of alternating nodes and packet classes). The first set F initially contains only the list $[n_0, t_1, n_1, true, \ldots, true, n_{m+1}]$ (line 3). In this list, t1 is equal to the Predicate {r.C.IPSrc, *, r.C.pSrc, *, *}, representing the largest traffic that satisfies the source component of r.C, while all the other traffics inside the list are set to true (i.e., the class of all packets).

Then, 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. For each node n_i of the path, the predicate b_i, representing the ingress traffic for that node in the current list, is split into the largest homogeneous subclasses of packets, by intersecting it with the forwarding domain ($\{I_a, I_d\}$ and the transformation domain ($\{D_{ij}\}$) of the node. Then, the corresponding function T is applied to each traffic that matches D_{ij}. For each partition of the Predicate b_i, a new list is generated and added to the result set of Maximal Flows. In these

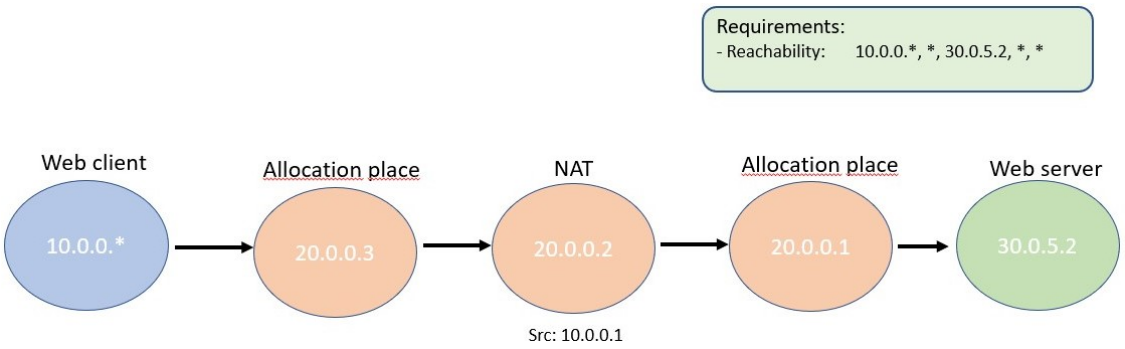
formulas, the operator + means list concatenation. Note that, for the packet filters in the APs (i.e., those whose configuration must be decided by the MaxSMT solver), I_a and I_d at the moment are unknown, because their configuration is not yet decided. For those nodes, I_a and I_d, are set respectively to true and false, i.e., no splitting occurs.

When the traffic arrives to the destination node, it is put in intersection, and therefore restricted, with the predicate representing the destination components of the requirement (i.e., {*, r.C.IPDst, *, r.C.pDst, r.C.tProto}). Starting from the flows computed in the forward traversal, a backward traversal is executed, which computes a new set of lists F'. F' is initialized to contain each element of F, with its last traffic restricted to the largest traffic that satisfy the destination components of r.C (line 8). So, with the backward traversal (lines 9-10), each predicate representing the ingress traffic of a node is restricted by propagating the restricted versions backwards.

The procedure stops when, after the last iteration, the flows in F and F' are the same. If not, a new iteration starts with F initially containing the flow present in F' at the end of the previous iteration.

8.2 Example

Let us consider this simple example that includes a sub-net of clients (10.0.0.*, corresponding to 10.0.0.0/24), a single transformer (NAT in 20.0.0.2), two allocation places and one web server. The requirement requires reachability between the sub-net of clients and the server. NAT performs Shadowing only on the IP address 10.0.0.1.



First, let us compute the input transformation classes for the NAT.

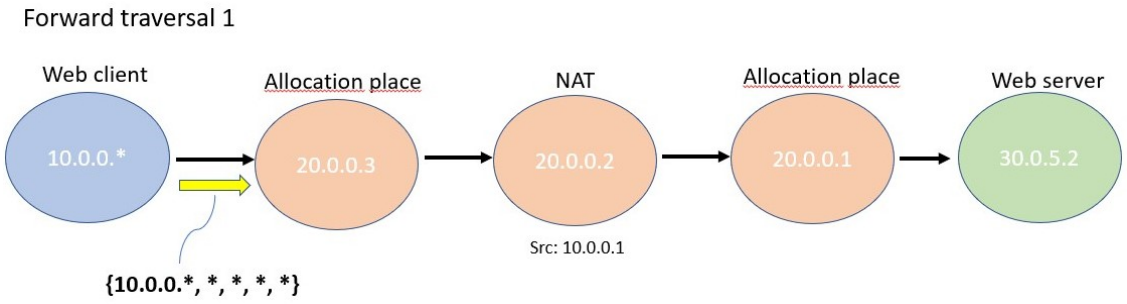
- D_1 = {10.0.0.1, *, !10.0.0.1 ∧ !20.0.0.2, *, *}
- D_2 = {!10.0.0.1 ∧ !20.0.0.1, *, 20.0.0.1, *, *}
- $D_3 = D_{31} \cup D_{32}$ = {10.0.0.1, *, 10.0.0.1, *, *} ∪ {!10.0.0.1, *, !10.0.0.1, *}

D_1 represents the Predicate matching the input class for the Shadowing operation, it has a private source address and a public destination address (different

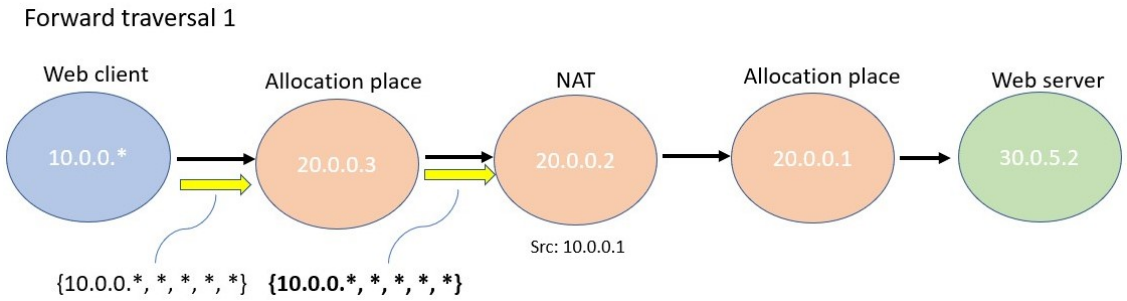
from the address of the NAT, 20.0.0.2). D_2 represents the Predicate matching the input class for the Reconversion operation, it has a public source address, and a destination address equals to the IP address of the NAT. D_3, instead, represents all Predicates not included in D_1 and D_2. In particular, in D_{31} we collect all the Predicates having both IPSrc and IPDst as private addresses, and in D_{32} all the Predicates having both IPsrc and IPDst as public addresses.

Now, let us compute all the paths that satisfy the requirement, that are the paths whose endpoints have IP address respectively equal to the IPSrc and IPDst expressed in the condition of the requirement. In this example, there exists a unique path represented by the list p = [10.0.0.*, 20.0.0.3, 20.0.0.2, 20.0.0.1, 30.0.5.2].

Now let us run Algorithm 5



Source traffic exiting from the source node is generated. It matches with the Predicate that has IPSrc and pSrc expressed by the condition of the requirement. For now, we do not yet consider the destination components of the requirement.

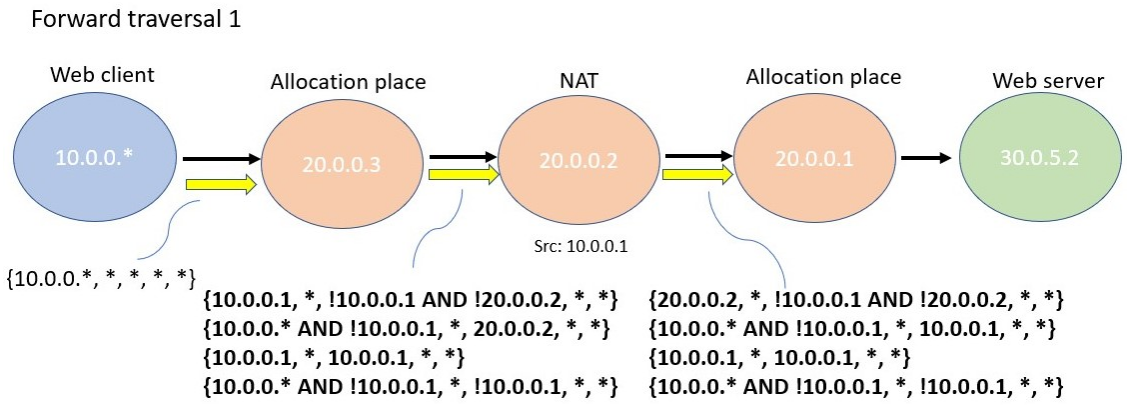


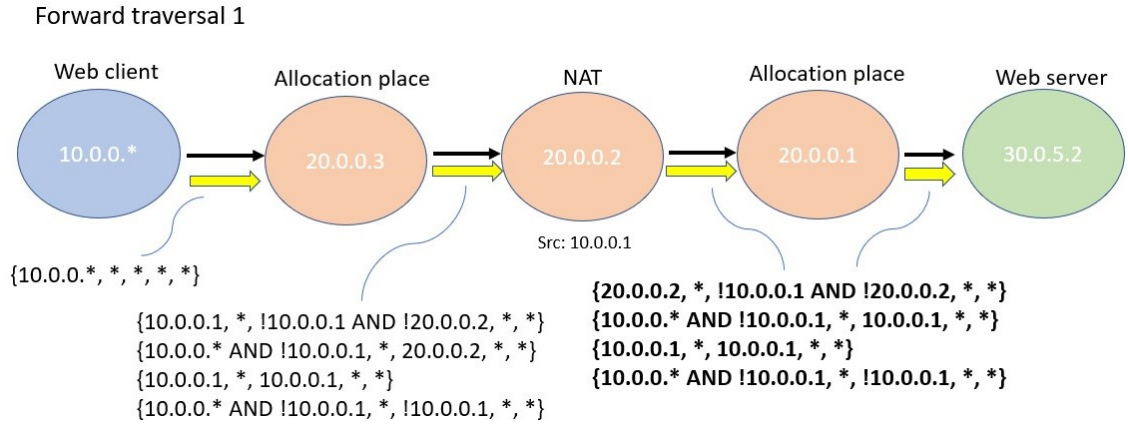
Once the allocation place is reached, the traffic is simply forwarded. As mentioned before, firewalls present in the allocation places are not yet configured, so their domains I_a and I_d are respectively equal to true and false. Therefore, in this phase, all the APs simply forward all the packets arriving them in input.

The predicate {10.0.0.*, *, *, *} arrives in input to the NAT. It is put in intersection with the various transformation domains of the NAT. In particular:

- {10.0.0.*, *, *, *} AND D_1 = {10.0.0.1, *, !10.0.0.1 ∧ !20.0.0.2, *, *}, followed by the Shadowing transformation that will modify the outgoing predicate in {20.0.0.2, *, !10.0.0.1 ∧ !20.0.0.2, *, *},

- {10.0.0.*, *, *, *} AND D_2 = {10.0.0.* ∧ !10.0.0.1, *, 20.0.0.2, *, *}, followed by the Reconversion transformation that will modify the outgoing predicate in {10.0.0.* ∧ !10.0.0.1, *, 10.0.0.1, *, *}
- {10.0.0.*, *, *, *} AND D_{31} = {10.0.0.1, *, 10.0.0.1, *, *}, simply forwarded
- {10.0.0.*, *, *, *} AND D_{32} = {10.0.0.* ∧ !10.0.0.1, *, !10.0.0.1, *, *}, simply forwarded



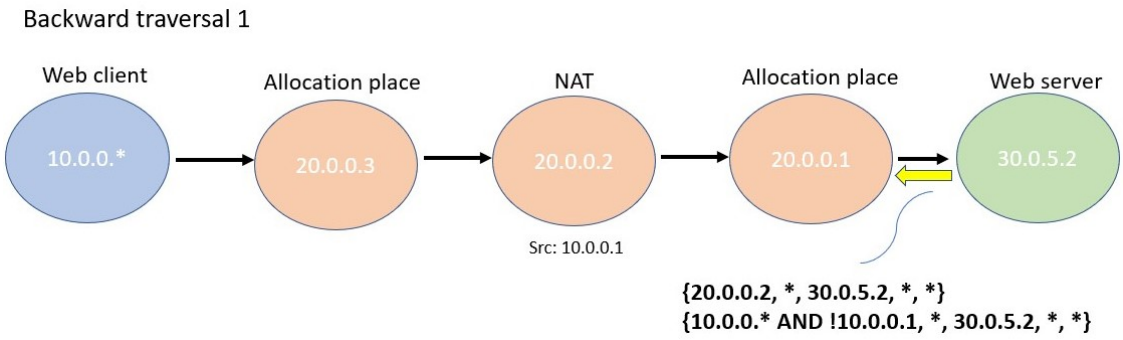


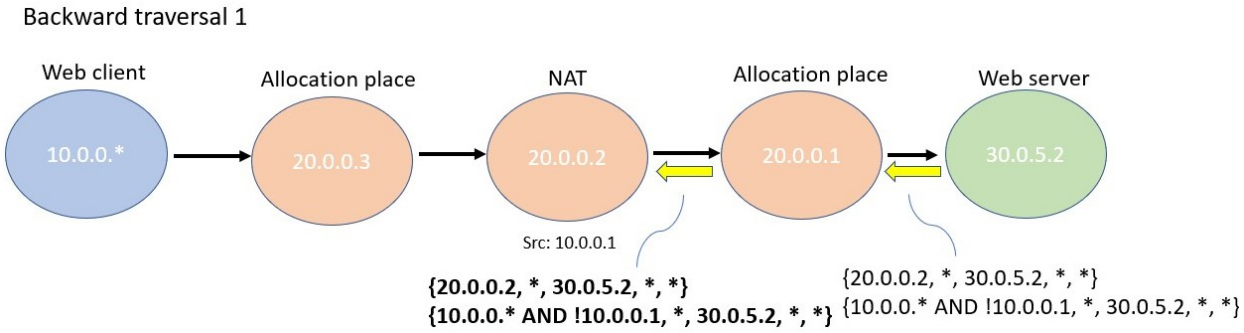
The various Predicates arrive to the second AP, they are simply forwarded and finally they reach the destination node. Here, they are put in intersection with the Predicate representing the destination components of the requirement, in our case with {*, *, 30.0.5.2, *, *}. For those whose intersection does not give a null result, the backward traversal begins. In particular,

- {20.0.0.2, * !10.0.0.1 ∧ !20.0.0.2, *, *} AND {*, *, 30.0.5.2, *, *} = {20.0.0.2, *, 30.0.5.2, *, *}
- {10.0.0.* ∧ !10.0.0.1, *, 10.0.0.1, *, *} AND {*, *, 30.0.5.2, *, *} = null. The Predicate is then discarded.
- {10.0.0.1, *, 10.0.0.1,*, *} AND {*, *, 30.0.5.2, *, *} = null. Also this Predicate is then discarded.

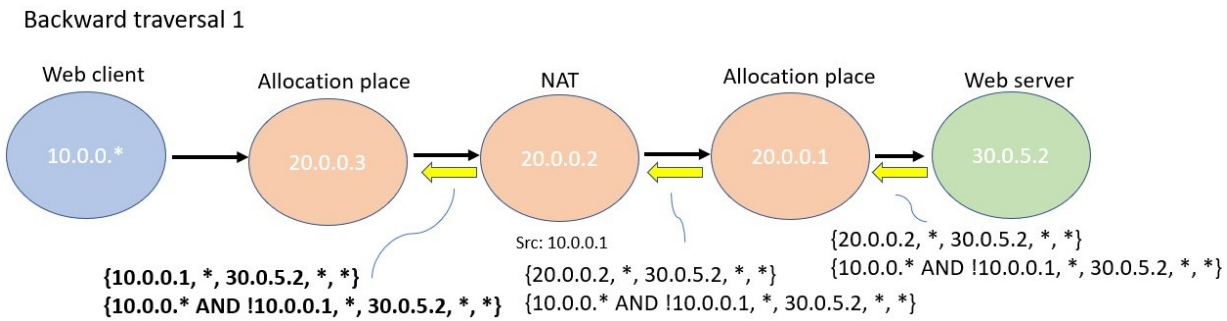
- {10.0.0.* ∧ !10.0.0.1, *, !10.0.0.1, *, *} AND {*, *, 30.0.5.2, *, *} = {10.0.0.* ∧ !10.0.0.1, *, 30.0.5.2, *, *}

Only two Predicates have a not null intersection with the destination Predicate and, therefore, they are propagated backward.





The various Predicates arrive at the NAT, the T^{-1} transformation is then applied. It is the inverse transformation to the one applied during the forward traversal.



Forward traversal 2 + Backward traversal 2 = the iteration is repeated a second time. During these two new traversals, the two remaining Predicates are not affected by any new transformation by all the nodes on the path. So, at the end of the second iteration, the Algorithm stops and returns the list containing the final Maximal Flows.

The two resulting Maximal Flows are lists of alternating nodes and Predicates:

1. [10.0.0.*, **{10.0.0.1, *, 30.0.5.2, *, *}**, 20.0.0.3, **{10.0.0.1, *, 30.0.5.2, *, *}**, 20.0.0.2, **{20.0.0.2, *, 30.0.5.2, *, *}**, 20.0.0.1, **{20.0.0.2, *, 30.0.5.2, *, *}**, 30.0.5.2]

2. [10.0.0.*, **{10.0.0.* ∧ !10.0.0.1, *, 30.0.5.2, *, *}**, 20.0.0.3, **{10.0.0.* ∧ !10.0.0.1, *, 30.0.5.2, *, *}**, 20.0.0.2, **{10.0.0.* ∧ !10.0.0.1, *, 30.0.5.2, *, *}**, 20.0.0.1, **{10.0.0.* ∧ !10.0.0.1, *, 30.0.5.2, *, *}**, 30.0.5.2]

8.3 Advantages

- This Algorithm is much faster than the two used for computing Atomic Flows. No initial computation time (to compute the set of Atomic Predicates) is required and therefore performance is much better.

8.4 Disadvantages

- It cannot be ensured that the Predicates generated by each Maximal Flows are minimal and disjoint. Specifically, a Predicate computed for one requirement might intersect with a Predicate computed for another requirement. So, for this reason, they cannot be assigned a unique integer, loosing all the advantages of working with integers rather than with Java classes.

- Since we can no longer identify the Predicates with integers, we should consider a class variable built ad hoc to give in input to z3. This variable, representing the Predicate class, is modelled with 13 fields: 4 integers representing the four bytes of the source IP address, 4 integers representing the four bytes of the destination IP address, 2 integers representing the range (min, max) of source ports, 2 integers representing the range of destination ports and a String representing the L4Prototype. So, as we can see, there are many more variables to give in input to z3, and therefore it is required much more time to solve the MaxSMT problem.

8.5 Maximal Flows VS Atomic Flows, introduction

The advantages and disadvantages of this approach are complementary to those analysed for the Atomic Flows approach. With Maximal Flows we have a very low flows generation time, but a higher time required to solve the MaxSMT problem. With Atomic Flows, on the contrary, we have a relatively high flows generation time, but a corresponding relatively low time taken by z3 (because it works with integer variables). The real challenge between the two approaches lies here: can the initial time taken to compute the set of all atomic predicates compensate and then bring sufficient advantages to z3? A final complete analysis is done in Chapter 10.

Chapter 9

Tests on Maximal Flows

9.1 Test parameters

The same tests presented in the Chapter 7 have been performed. The only difference is in the true/false parameter indicating whether the configured firewall rules were to be randomly generated on the basis of the requirements or not. Here, this parameter is not longer necessary. According to some preliminary test, in fact, the value of this parameter did not affect the execution time of Algorithm 5. The configurable parameters are then: **number of requirements**, **number of web clients and web servers**, **number of allocation places**, **number of NATs**, **number of FWs**, **number of sources within each NAT**, **number of configured rules within an already existing firewall**, **percentage** of requirements with information also on port numbers and protocol type.

Also in this case, there are two approaches chosen for building the tests:

- The first kind of tests consist of selecting a "basic configuration" for all the parameters, and then increase the value of one parameter at a time while keeping all the other constants, in order to evaluate how much that parameter affects the times. Here too, we have chosen for the "basic configuration" the case with "**100 REQ, 100 WC, 100 WS, 100 AP, 25 NAT, 25 FW, 10 NATSrcs, 10 FWRules**".
- The second kind of tests consist of increasing simultaneously the values of all the parameters, in order to simulate a progressive enlargement of the network.

9.2 Analysis of test results

Let us see how single parameters affect the total execution time.

Number of requirements. As we can see from the graph below (Figure 9.1), the number of requirements does not particularly affect the total execution time of the Algorithm. It goes from 0.13 seconds (for the case with 100 requirements) to 10.6 seconds (for the case with 10 thousand of requirements). Although the number

of requirements has increased a lot, times are kept low. Let us remember that with the Atomic Predicates approach, the average times with FALSE configuration were 4.7 seconds (with 100 requirements) and 86 seconds (with 10 thousand of requirements). Here, the increase in time is due to the fact that there are more requirements for which the corresponding Maximal Flows must be computed. So, Algorithm 5 is run more times, once for each requirement.

Number of web clients and web servers, number of allocation places. As with the Atomic Predicates approach, here too the total number of WCs, WSs and APs is irrelevant. It does not involve any increase in times, which are always kept very low for all the cases (less than one second).

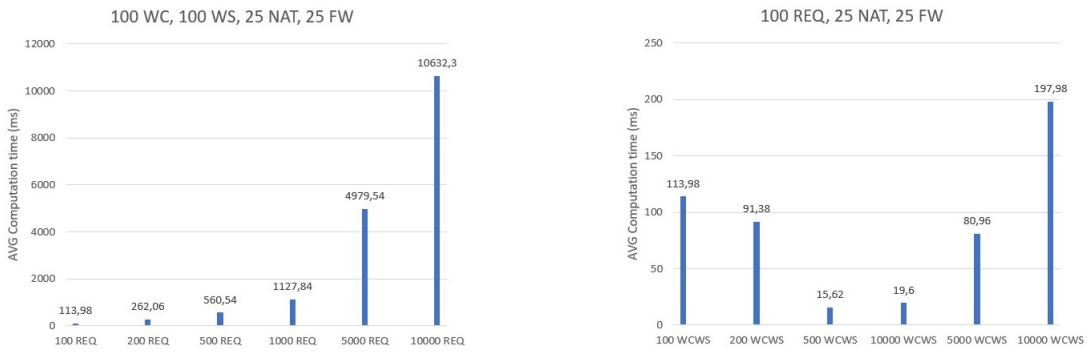


Figure 9.1: Computation time VS number of requirements, Computation time VS number of web clients and web servers

Number of NATs. The total number of NATs is a determining parameter in the increase of times. Looking at Algorithm 5 and at the Example described above, we can see that each NAT, and in general each transformer, increase the number of flows to be generated. Each flow entering a transformer, in fact, is split into multiple flows according to the various intersections the incoming predicate has with the transformation domains of the node. The more NATs there are, the more intersection and therefore the more flows will be generated.

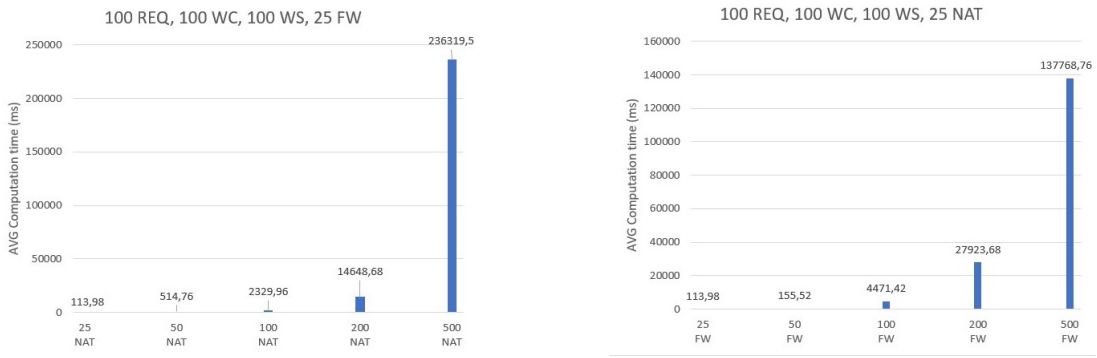


Figure 9.2: Computation time VS number of NATs, Computation time VS number of Firewalls

Number of firewalls. As well as the number of NATs, the total number of firewalls determines an increase in times, even if less significant. Let us remember that, in line 5 of Algorithm 5, the Predicate entering a node is placed in intersection

with its forwarding behaviour, I_a and I_d. In case of firewalls, I_a and I_d consist of the set of all Predicates expressing the condition of the rules configured inside that firewall. Each existing intersection with one of these Predicates, generates a new Maximal Flow. Therefore, it determines an increase in times required by Algorithm 5 to complete.

Number of sources within each NAT. They do not particularly affect times, all the tests considered took less than one second to complete. The real discriminating factor is the number of NATs present in the network, not the number of sources present within them.

Number of rules configured within each existing firewall. Times increase slightly. There are mainly two reasons for this. The first reason is because the more rules configured inside each firewall, the more time to compute the set I_a and I_d because there are more rules to process. However, this time has relatively little effect on performance. The second and most significative reason is the fact that more rules are configured inside each firewall, more predicates will be present inside I_a and I_d, and so a greater probability of intersection for the predicate entering the node. Remember that for each intersection, a new Maximal Flow is generated.

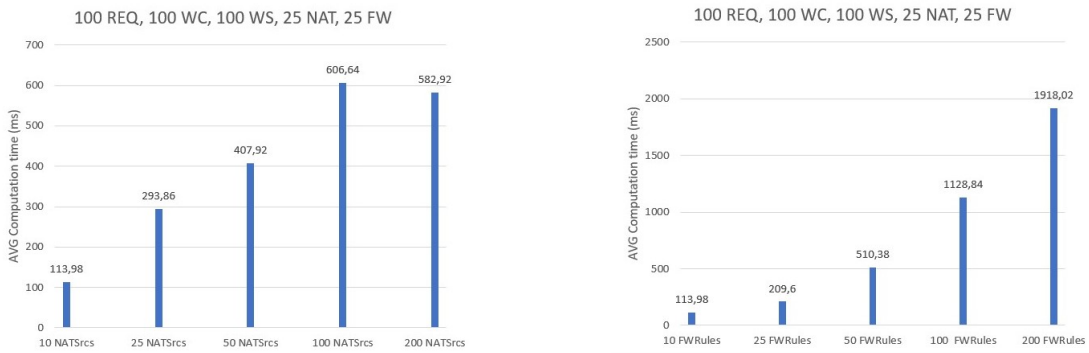


Figure 9.3: Computation time VS number of NAT sources, Computation time VS number of firewall rules

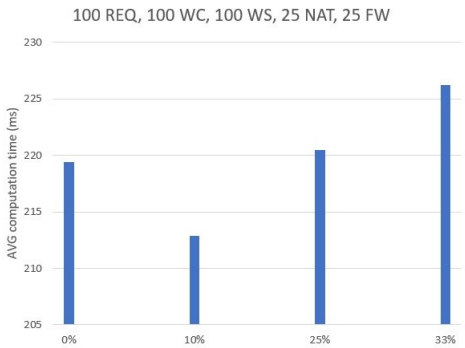


Figure 9.4: Computation time VS percentage of requirements with information on ports and protocol type

Percentage of requirements with information also on port numbers and protocol type. With the Maximal Flows approach, this parameter does not affect times. All the transformers considered in our examples, work mostly by

changing only the IP addresses of packets, so the information on ports and protocol type is simply forwarded without being modified, The situation could be different if we introduce in the network transformers that also modify port numbers and protocol type.

Progression tests

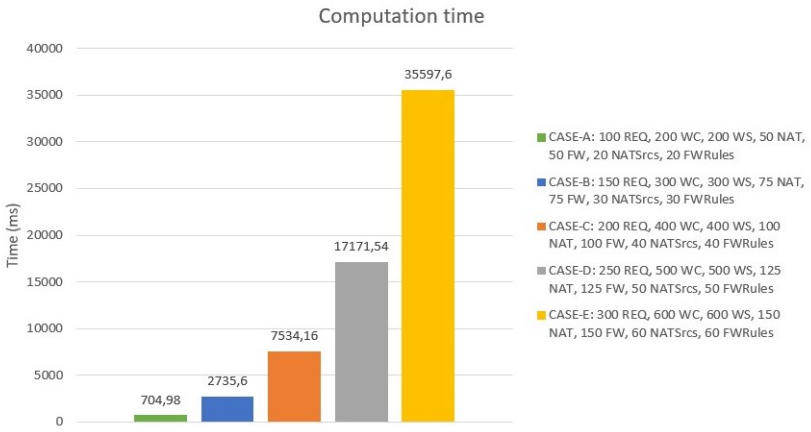


Figure 9.5: Computation time VS progression tests

.

Chapter 10

Atomic Flows VS Maximal Flows

This chapter deals with the final analysis about the comparison between the two approaches. In particular, the results obtained by solving the entire Refinement problem will be described. As described in Chapter 3, the work done by the Refinement tools can be mainly divided into two phases. The first phase consists of computing all the Traffic Flows related to the requirements expressed in input by the user. We have analyzed in detail the time taken for the computation of these flows for both the two approaches in Chapters 7 and 9. The second phase instead consists of giving the flows computed in the previous phase in input to a MaxSMT solver - z3 in our case - , in order to find the optimal allocation and configuration of the security mechanisms necessary to satisfy the expressed requirements. As anticipated at the end of Chapter 8, this phase is critical not only for the number of traffic flows that the solver could receive in input, but also for the type of class used to represent them. With the approach of Maximal Flows, the solver receives in input a complex class representing the five main fields of the IP header, and models it using only the simple variable types supported by z3: 4 integers to represent the four bytes of the source IP, 4 integers for the destination IP, 2 integers to represent the range of source ports, 2 integers for the range of destination ports and a string with the concatenation of protocol types, giving a total of 13 variables to express each predicate within each Maximal Flow. With the approach of Atomic Flows, on the other hand, the solver receives in input simple integers, which are the identifiers of the predicates within each Atomic Flow, unique and disjoint within the network. And this is the main advantage of Atomic Flows over Maximal Flows.

Concerning the number of generated flows, with the Atomic Flows approach a higher number are generated. And this can be easily understood since with Atomic Flows we try to slit each flow into minimal and simpler traffic flows (reaching a higher level of granularity), while with Maximal Flows we try to aggregate as much as possible the various flows into flows called Maximal that are equally expressive. And this is therefore a point in favor of the second approach.

The last consideration that could play a relevant role is the time taken to compute the traffic flows (phase 1). In the case of Maximal Flows, we have seen that the computation is immediate, very short time is lost to carry out this phase since the process of generating the flows is a simple recursive function (5), mostly parallelizable. While for Atomic Flows, it is a particularly influential time. It corresponds

to the initial time spent to compute the global set of atomic predicates, starting from the "interesting" predicates of the network (as described in Algorithm 3), and the subsequent recursive function (Algorithm 4), which, in turn, computes the corresponding Atomic Flows. This initial time spent on the computation of atomic predicates could be a big disadvantage.

Consequently, we could summarized as follows:

- Phase 1 of the Refinement problem - i.e. computation of traffic flows - computationally expensive for the approach with Atomic Flows, while irrelevant for the approach with Maximal Flows
- Phase 2 of the Refinement problem - i.e. solving the MaxSMT problem - much more expensive for Maximal Flows.

As written in the last section of Chapter 8, the real challenge between the two approaches is precisely this: to evaluate how much weight phase 1 has compared to phase 2, with respect to the total time spent for the Refinement problem. If phase 1 has a lighter weight than phase 2 (that means in proportion it takes less time to be solved for both the two approaches), then it is worth spending some initial time to compute the Atomic Predicates and assign them an integer identifier so that phase 2 can be solved in a more flexible and faster way. On the contrary, if phase 1 has a heavier weight than phase 2, then the initial time spent could prove inconvenient.

10.1 Tests execution

What we have obtained by analyzing the results is mainly that phase 2 of the Refinement problem (the MaxSMT phase) has a much greater impact than phase 1, and consequently the Atomic Flows approach is more advantageous than the one with Maximal Flows. We have analyzed that the initial time spent to compute the set of Atomic Predicates and the corresponding Atomic Flows, for the analyzed networks, is not much greater than the one spent to compute the Maximal Flows, but it brings enough advantages to make phase 2 significantly faster.

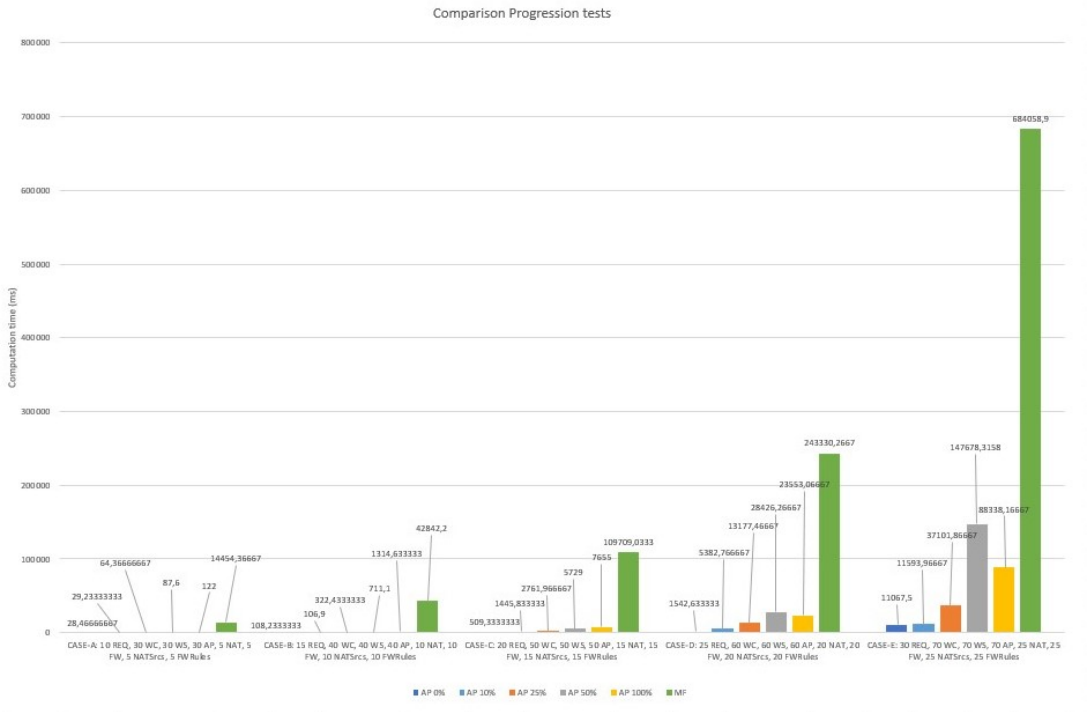
The parameters considered for the tests are the same described in Chapter 7 and Chapter 9. We will analyse a series of tests in which the network and the values of all the parameters are progressively increased step by step and we will describe how much time the two phases of the Refinement problem take to be solved, in order to evaluate the weight of each phase with respect to the total time.

Since the percentage of requirements that bring information also on ports and protocol type is particularly critical for the Atomic Flows approach (as we have seen in the last section of Chapter 7), we analysed this approach in five different scenarios: with 0%, 10%, 25%, 50% and 100% of requirements complete of the information on ports and protocol types. Concerning the Maximal Flows approach instead, we have seen in Chapter 9 that this parameter does not affect performances, so for this approach we consider the case with 0%.

10.2 Analysis of test results

10.2.1 Final results

The final results with the comparison of the two approaches can be summarized by the following figure.



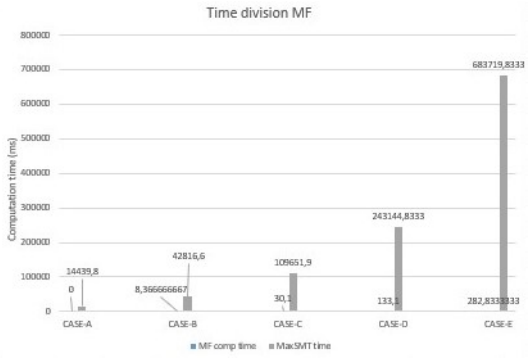
As we can see, the approach with Maximal Flows takes much more time than all the other five scenarios considered for the approach with Atomic Flows. Therefore the latter seems to be the most convenient approach, despite the percentage of requirements with information also on ports and protocol types is a very influential parameter.

The figure seems to show a behavior that, at first sight, might be strange: the case with AP 100% takes less time to complete than the case with AP 50%. In reality, considering how Atomic Predicates and Atomic Flows are built, we can understand the causes of this behavior. They will be explained well in the next section, which will analyze the time difference between the two phases of the Refinement problem for all six considered cases.

10.2.2 Time division between the phases

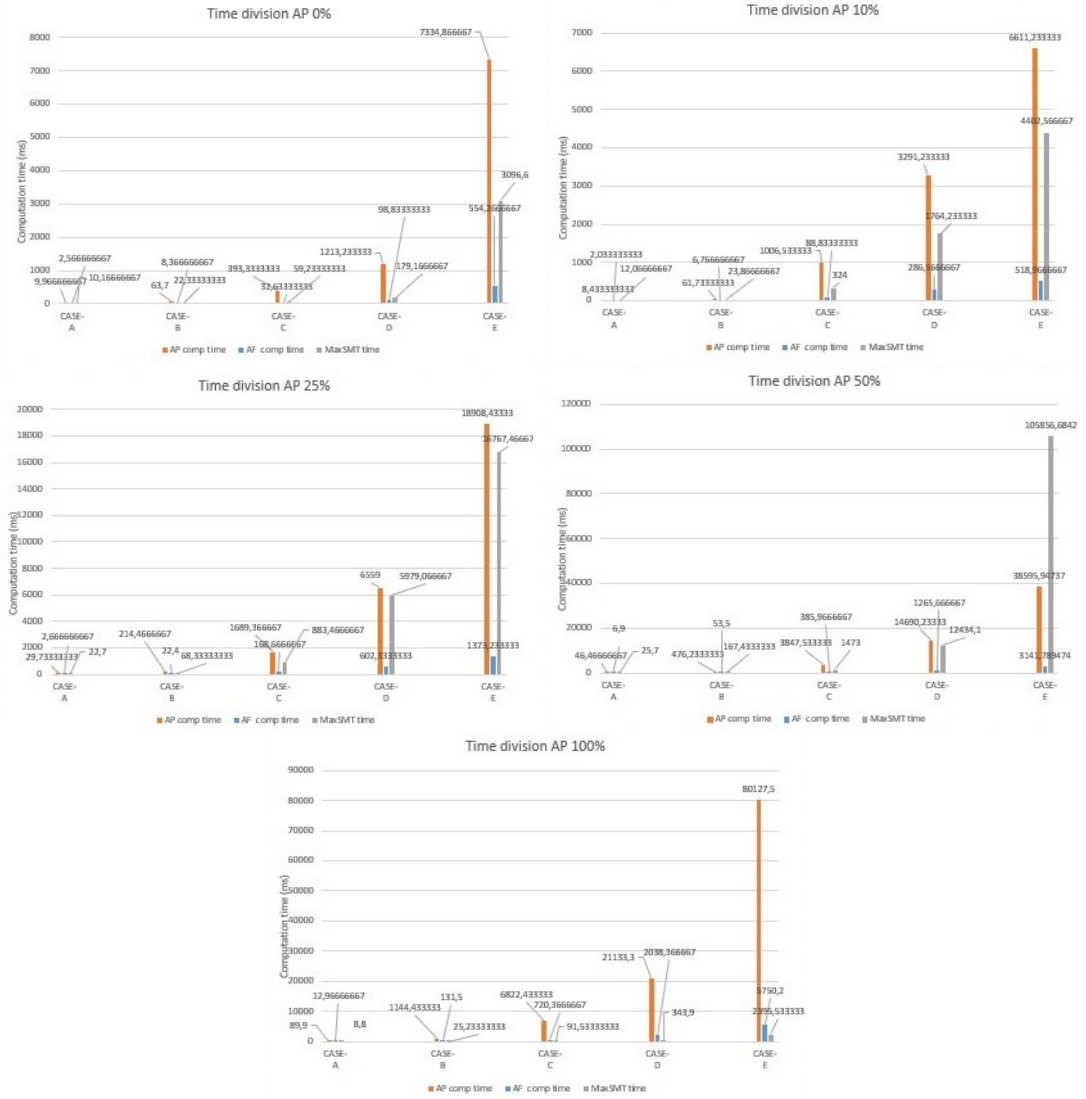
Maximal Flows approach

As we can see from the following figure, the time taken with the Maximal Flows approach is almost entirely to be attributed to the resolution of the MaxSMT problem. Only few milliseconds are spent for the computation of the Maximal Flows.



Atomic Flows approach

More interesting is to analyze the results obtained with the approach of Atomic Flows.



Time is more homogeneously divided between time to generate the Atomic predicates and time to solve the MaxSMT problem. The recursive function to generate Atomic Flows starting from Atomic Predicates, instead, is much immediate.

We can state that time to solve the MaxSMT problem is directly related to the number of predicates the solver receives in input. In the cases with 0% and 10%, the resolution of the MaxSMT problem is very fast and the most of the time is taken to compute the Atomic Predicates. Increasing the percentage, more time is taken for the computation of Atomic Predicates (as we have seen in the last section of Chapter 7) but also to solve the MaxSMT problem, because it will receive in input more predicates and flows. In the case with 50% the time taken for the resolution of the MaxSMT problem becomes predominant. The time taken to solve the case with 100% is an expected exception.

Let us see a clarifying **example**.

Let us consider a case with 10 requirements and 50% of requirements with information also on source port (we consider only source port for simplicity but the issue could be extended also to destination port and protocol type).

In this case, we will have 5 requirements with a specific source port (i.e., port a, b, c, d, e) and 5 requirements that express the information on source port through the wildcard (that means, any source port value is allowed).

The resulting set of Atomic Predicates could be, in turn, divided into 6 main groups. The first group for Atomic Predicates with source port equals to a, the second with source port equals to b, the third with source port equals to c etc. The sixth group with source port with a value different from a, b, c, d, e.

Now let us consider the resulting Atomic Flows, built on this set of Atomic Predicates. Each one of the requirements that bring information also on source port will be defined by a single Atomic Flows starting with the Predicate {source IP x, dest IP y, source port $a/b/c/d/e$, dest port *}. So in this case, only one Predicate will be included in the set B_0 for Algorithm 4, that is precisely the one with the specific value for the source port expressed by the requirement. For the other requirements, which does not have information on source port (i.e., all source ports allowed), the starting set B_0 will include 6 predicates: regardless of the other information present in the predicate, we will have the predicate with source port equals to a, the predicate with source port equals to b, the predicate with source port equals to c etc. The last one corresponds with the predicate having source port value different from a, b, c, d, e. The disjunction of these six predicates will represent any predicate related to the requirement, with all possible values for the source port. Therefore, each one of the requirements with no information on source port will generate at least 6 different Atomic Flows.

The total number of generated flows would be = 5*1 + 5*6 = 35 Atomic Flows to give in input to the MaxSMT solver (minimum number).

A case with 20% of requirements with information also on source port (2 requirements against 10 considered) will generate 3 different groups of Atomic Predicates grouped by source port. The total number of atomic flows will be 2*1 + 8*3 = 26 Atomic Flows.

A case with 90% of requirements with information also on source port (9 requirements against 10 considered) will generate 9*1 + 1*10 = 19 Atomic Flows.

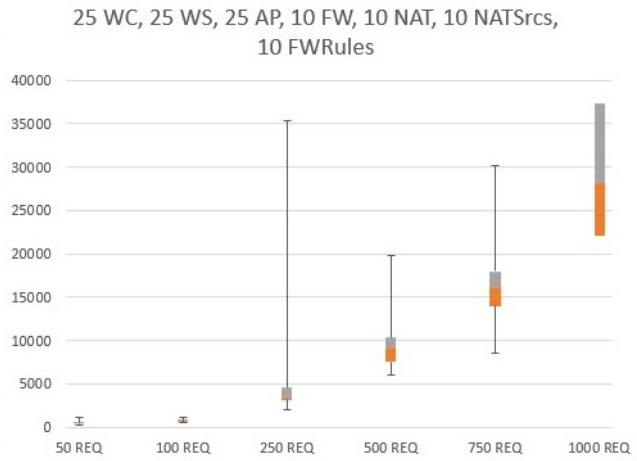
A case with 100% of requirements with information also on source port (10 requirements against 10 considered) will generate 10*1 = 10 Atomic Flows.

And this is the reason why the case with 100% takes much less time to solve the MaxSMT problem, that is because the total number of Atomic Flows to give in input to the solver will be lower.

The time to generate the atomic predicates will always increase in any case with the increase of the percentage of information on ports and protocol types, because more information must be taken into consideration and therefore a greater number of atomic predicates will be generated, as we have seen in last section of Chapter 7 and as we can see considering the red lines in the last graphs. Instead, the number of generated Atomic Flows to give in input to the solver, and consequently the time to solve the MaxSMT problem, does not increase proportionally with the increase in the percentage of information on ports and protocol type, but, as we have seen, arrived at a certain percentage there is a turnaround and it begins to decrease. This is why, in the first graph shown in this Chapter, the total time was lower in the 100% case than in the 50% case.

10.2.3 Stressing the Atomic Predicates approach

We have also tried to stress the approach with Atomic Flows considering a medium/large size network and a very high number of requirements. The network consisted of 25 Web clients, 25 Web servers, 25 allocation places, 10 NAT, 10 Firewalls, 10 NAT sources and 10 rules configured in each firewall. For these tests, we have considered the case with 0% of requirements with information also on ports and protocol types, so actually the Refinement problem becomes a Reachability problem, having only information on IP sources and IP destinations. We have seen that the approach that makes use of Atomic Flows performs very well and its behavior can be summarized in the following graph.



Chapter 11

Conclusions

With this Chapter, we complete the work done for this thesis. Let us try to summarize what has been done.

First of all, we analyzed what is the state-of-the art in traffic flows and network functions modelling, focusing on its importance in solving, between the many, the Refinement problem. This problem aims to find the optimal allocation and configuration of security mechanisms (such as firewalls) on the basis of requirements, that are expressed by the network designer in a high-level language. We then moved on to propose a new model to represent predicates that was Java compatible and therefore usable within the VEREFOO framework, which aims to solve the aforementioned Refinement problem. We called this new class Predicate.

At this point we studied and proposed two new models to describe traffic flows and network functions, fully complementary and alternative, one based on Atomic Flows and one based on Maximal Flows, which make use of the class Predicate to describe the predicates of the network. Each one of these models must be able to represent the behavior of the network, in order to predict how any packet that is introduced into the network is treated and possibly modified by the various nodes it crosses. In the Appendix of this thesis a possible implementation of the main algorithms for the two proposed approaches, described in Chapter 6 and Chapter 8, is shown and that is also the implementation used within VEREFOO. Having defined and implemented each model, it was therefore necessary to test it, to evaluate how efficient it was in the context of real network scenarios, and to study its behavior varying not only the size of the network, but also the value of each single parameter that we have considered for the tests (number of nodes in the network, number of transformers, number of requirements expressed, etc.). The evaluation of the tests must take into account the total time taken by each approach to solve the Refinement problem, which can be mainly divided into two phases: the first phase which computes all the possible classes of traffic that can travel through the network, concerning the requirements expresssed by the user, and the second one that solves a weighted and partial MaxSMT problem to find the optimal allocation and configuration of the firewalls, in order to satisfy the above requirements. As we showed in Chapter 10, we got interesting results with the Atomic Flow method.

This work done lends itself very well to being extended. Future works may

concern both the definition of new models to describe other network functions (for now we have considered only firewalls and nats but both approaches are well suited to describe any network function), and optimization techniques to further improve the obtained performance.

And this is all, thank you for the attention.

Bibliography

[1] 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. [Online]. Available: https://doi.org/10.1109/CCCS.2019.8888130

[2] I. Pedone, A. Lioy, and F. Valenza, "Towards an efficient management and orchestration framework for virtual network security functions," *Secur. Commun. Networks*, vol. 2019, pp. 2 425 983:1–2 425 983:11, 2019. [Online]. Available: https://doi.org/10.1155/2019/2425983

[3] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110402

[4] ——, "Introducing programmability and automation in the synthesis of virtual firewall rules," in *6th IEEE Conference on Network Softwarization, NetSoft 2020, Ghent, Belgium, June 29 - July 3, 2020*, F. D. Turck, P. Chemouil, T. Wauters, M. F. Zhani, W. Cerroni, R. Pasquini, and Z. Zhu, Eds. IEEE, 2020, pp. 473–478. [Online]. Available: https://doi.org/10.1109/NetSoft48620.2020.9165434

[5] E. Karafili and F. Valenza, "Automatic firewalls' configuration using argumentation reasoning," in *Emerging Technologies for Authorization and Authentication - Third International Workshop, ETAA 2020, Guildford, UK, September 18, 2020, Proceedings*, ser. Lecture Notes in Computer Science, A. Saracino and P. Mori, Eds., vol. 12515. Springer, 2020, pp. 124–140. [Online]. Available: https://doi.org/10.1007/978-3-030-64455-0_8

[6] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 113–126. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[7] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 99–111. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[8] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow:

Verifying network-wide invariants in real time," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 15–27. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[9] F. Valenza, "Modelling and analysis of network security policies," Ph.D. dissertation, Doctoral Dissertation Doctoral Program in Computer Engineering (29th cycle . . . , 2017.

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

[11] L. Durante, L. Seno, F. Valenza, and A. Valenzano, "A model for the analysis of security policies in service function chains," in *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–6.

[12] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2016.

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

[14] E. Wong, "Validating network security policies via static analysis of router acl configuration," 2006. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA462558

[15] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, "Improving the formal verification of reachability policies in virtualized networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 713–728, 2021.

[16] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano, "A comprehensive approach to the automatic refinement and verification of access control policies," *Comput. Secur.*, vol. 80, pp. 186–199, 2019. [Online]. Available: https://doi.org/10.1016/j.cose.2018.09.013

[17] ——, "A comprehensive approach to the automatic refinement and verification of access control policies," *Computers & Security*, vol. 80, pp. 186–199, 2019.

Appendices

Appendix A

Implementation Atomic Flows

This appendix shows a possible implementation in Java of the main functions for the computation of Atomic Predicates and Atomic Flows, described by Algorithms 2, 3 e 4. The following described implementation is the one used within the framework *VEREFOO*.

All the functions shown in this Appendix make use of the class Predicate and the operations over Predicates described in the Chapter 4.

A.1 Algorithm 2: Atomic Predicates computation

```
public List<Predicate> computeAtomicPredicates(List<Predicate> atomicPredicates,
        List<Predicate> predicates){
    List<Predicate> newAtomicPredicates = new ArrayList<>();
    Predicate first = null;
    List<Predicate> firstNeg = null;
    int count = -1;

    for(Predicate sp: predicates) {
        //If sp is the first predicate to transform and atomicPredicates is empty
        if(atomicPredicates.isEmpty() && count == -1) {
            first = sp;
            firstNeg = neg(sp);
            count = 1;
        }
        else if(count == 1) {
            //There is already a predicate in the list, and this is the second
            Predicate sp1 = computeIntersection(first, sp);
            if(sp1 != null) atomicPredicates.add(sp1);

            for(Predicate s: firstNeg) {
                Predicate sp2 = computeIntersection(s, sp);
                if(sp2 != null) atomicPredicates.add(sp2);
            }

            for(Predicate s: neg(sp)) {
                Predicate sp3 = computeIntersection(first,s);
                if(sp3 != null) atomicPredicates.add(sp3);
            }
```

```
            for(Predicate s1: neg(sp)) {
                for(Predicate s2: firstNeg) {
                    Predicate sp4 = computeIntersection(s1,s2);
                    if(sp4 != null) atomicPredicates.add(sp4);
                }
            }

            count = -1;
        } else {
            //there are already more then 2 predicates
            for(Predicate prevSp: atomicPredicates) {
                Predicate res1 = computeIntersection(prevSp, sp);
                if(res1 != null) newAtomicPredicates.add(res1);

                for(Predicate s: neg(sp)) {
                    Predicate res2 = computeIntersection(prevSp,s);
                    if(res2 != null) newAtomicPredicates.add(res2);
                }
            }
            atomicPredicates = new ArrayList<>(newAtomicPredicates);
            newAtomicPredicates = new ArrayList<>();
        }
    }
    if(count == 1) {
        firstNeg.add(first);
        return firstNeg;
    }
    return atomicPredicates;
}
```

This function receives in input a set of already computed Atomic Predicates (*atomicPredicates*) and a set of Predicates (*predicates*), not yet atomic, to convert and add to the set. For each predicate within *predicates*, the corresponding list representing its negation is computed through the function *neg* (??), according to lines 1-3 of Algorithm 2. At this point, the predicate itself and each predicate representing its negation are placed, one at a time, in intersection with each of the predicates already present in the list *atomicPredicates*. If the intersection exists, then, it is added to the list of resulting atomic predicates, according to lines 4-6 of the Algorithm. At the end, the resulting updated list of atomic predicates is returned.

A.2 Algorithm 3: "Interesting" Predicates and corresponding Atomic Predicates computation

This is the algorithm in charge of collecting the "interesting" predicates for the network, on the basis of the requirements specified in input, and subsequently converting them into the corresponding set of atomic predicates, using the function described above.

A.2.1 Interesting predicates for source and destination traffic of each requirement

For the first part of Algorithm 3 (lines 1-4), "interesting" predicates related to source and destination traffic for each requirement are computed.

```
private HashMap<Integer, Predicate> generateAtomicPredicateNew(){
    List<Predicate> predicates = new ArrayList<>();
    List<Predicate> atomicPredicates = new ArrayList<>();
    List<String> srcList = new ArrayList<>();
    List<String> dstList = new ArrayList<>();
    List<String> srcPList = new ArrayList<>();
    List<String> dstPList = new ArrayList<>();
    List<L4ProtocolTypes> dstProtoList = new ArrayList<>();

    //Generate predicates representing source and predicates representing
    //destination of each requirement
    for(SecurityRequirement sr : securityRequirements.values()) {
        Property property = sr.getOriginalProperty();
        String IPSrc = property.getSrc();
        String IPDst = property.getDst();
        String pSrc = property.getSrcPort() != null &&  !property.getSrcPort().equals("null")
                ? property.getSrcPort() : "*";
        String pDst = property.getDstPort() != null &&  !property.getDstPort().equals("null")
                ? property.getDstPort() : "*";
        L4ProtocolTypes proto = property.getLv4Proto() != null ?
                property.getLv4Proto() : L4ProtocolTypes.ANY;
        srcList.add("*"); dstList.add("*"); srcPList.add("*"); dstPList.add("*");
        dstProtoList.add(L4ProtocolTypes.ANY);

        //if we have already inserted this source into the list, we can skip it
        if(!srcList.contains(IPSrc) || !srcPList.contains(pSrc)) {
            if(!srcList.contains(IPSrc))
                srcList.add(IPSrc);
            else IPSrc = "*";
            if(!srcPList.contains(pSrc))
                srcPList.add(pSrc);
            else pSrc = "*";

            Predicate srcPredicate = new Predicate(IPSrc, false, "*", false,
                    pSrc, false, "*", false, L4ProtocolTypes.ANY);
            predicates.add(srcPredicate);
        }

        //if we have already inserted this destination into the list, we can skip it
        if(!dstList.contains(IPDst) || !dstPList.contains(pDst)
                || !dstProtoList.contains(proto)) {
            if(!dstList.contains(IPDst)) dstList.add(IPDst);
            else IPDst = "*";
            if(!dstPList.contains(pDst)) dstPList.add(pDst);
            else pDst = "*";
            if(!dstProtoList.contains(proto)) dstProtoList.add(proto);
            else proto = L4ProtocolTypes.ANY;

            Predicate dstPredicate = new Predicate("*", false, IPDst, false, "*",
                    false, pDst, false, proto);
            predicates.add(dstPredicate);
        }
    }
```

To avoid redundancy and, in this way, speed up the algorithm that will subsequently compute the corresponding set of atomic predicates, if a field of an interesting predicate is already present in *predicates*, thus it is not added anymore.

A.2.2 Interesting predicates for forwarding behaviour and transformation input domains

For the second part of Algorithm 3 (lines 5-6), "interesting" predicates related to the forwarding behaviour $\{I_a, I_d\}$ and to the transformation input domains of each node, encountered along the path of at least one requirement, are computed. For the moment, we only consider Firewalls and NATs.

Firewalls

The forwarding behaviour for pre-existing firewalls is computed on the basis of the conditions of the rules that are already configured within it. According to Algorithm 1, it is possible to convert an ACL into a set of corresponding predicates, and this is exactly what is done in our implementation.

```
if(node.getFunctionalType() == FunctionalTypes.FIREWALL) {

List<Predicate> allowedList = new ArrayList<>();
List<Predicate> deniedList = new ArrayList<>();

boolean deniedListChanged = false;
for(Elements rule: node.getConfiguration().getFirewall().getElements()) {
    if(rule.getAction().equals(ActionTypes.DENY)) {
        //deny <--- deny V rule-i
        deniedList.add(new Predicate(rule.getSource(), false, rule.getDestination(), false,
                rule.getSrcPort(), false, rule.getDstPort(), false, rule.getProtocol()));
        deniedListChanged = true;
    } else {
        //allowed <--- allowed V (rule-i AND !denied)
        Predicate toAdd = new Predicate(rule.getSource(), false, rule.getDestination(), false,
                rule.getSrcPort(), false, rule.getDstPort(), false, rule.getProtocol());
        List<Predicate> allowedToAdd =
                aputils.computeAllowedForRule(toAdd, deniedList, deniedListChanged);
        for(Predicate allow: allowedToAdd) {
            if(!aputils.isPredicateContainedIn(allow, allowedList))
                allowedList.add(allow);
        }
    }
}
//Check default action: if DENY do nothing
if(node.getConfiguration().getFirewall().getDefaultAction().equals(ActionTypes.ALLOW)) {
    Predicate toAdd = new Predicate("*", false, "*", false, "*",
            false, "*", false, L4ProtocolTypes.ANY);
    List<Predicate> allowedToAdd =
            aputils.computeAllowedForRule(toAdd, deniedList, deniedListChanged);
    for(Predicate allow: allowedToAdd) {
        if(!aputils.isPredicateContainedIn(allow, allowedList))
            allowedList.add(allow);
    }
}
```

For each rule configured within the firewall, if the rule has a deny action, then the corresponding predicate is computed and directly added into *deniedList*, which is the list containing the conditions for the rules having action equal to deny. If the rule, instead, has an allow action, the corresponding predicate is computed and added to *allowedList* only after removing the part of the condition that possibly intersects with an already scanned deny predicate (to respect the first-match criterion with which the rules are inserted inside firewalls).

Function *computeAllowedForRule*, which execute the logic operation ($rule_i \wedge \neg denied$), is implemented by the following piece of code.

```
List<Predicate> negDeniedRuleList;
public List<Predicate> computeAllowedForRule(Predicate toAdd,
        List<Predicate> deniedList, boolean deniedListChanged){
    List<Predicate> retList = new ArrayList<>();
    List<Predicate> tmpList = new ArrayList<>();
    retList.add(toAdd);

    if(deniedList.isEmpty()) return retList;

    for(Predicate deniedRule: deniedList) {
        //compute !denied
        if(deniedListChanged)
            negDeniedRuleList = neg(deniedRule);
        for(Predicate p1: retList) {
            for(Predicate p2: negDeniedRuleList) {
                Predicate res = computeIntersection(p1, p2);
                if(res != null) {
                    tmpList.add(res);
                }
            }
        }
        if(tmpList.isEmpty()) {
            //no intersection exists
            return new ArrayList<>();
        } else {
            retList = new ArrayList<>(tmpList);
            tmpList = new ArrayList<>();
        }
    }
    return retList;
}
```

At this point, all the predicates present inside *allowedList* and *deniedList* are added to *predicates*, which is the set of "interesting" predicates of the network computed so far.

NATs

Let us now compute the transformation input domains for the NATs. With the following piece of code, the input domains for Shadowing, Reconversion and the corresponding inverse functions are computed.

```
for(Node node: transformersNode.values()) {
    if(node.getFunctionalType() == FunctionalTypes.NAT) {
        //Compute list of shadowed and reconverted (only those related to requirements sources),
        //considering NAT source addresses list
        List<String> shadowedAddressesListSrc = new ArrayList<>();
        List<String> shadowedAddressesListDst = new ArrayList<>();
        for(String shadowedAddress: node.getConfiguration().getNat().getSource()) {
            for(String ips: srcList) {
                if(shadowedAddress.equals(ips)
                        || aputils.isIncludedIPString(shadowedAddress, ips)) {
                    shadowedAddressesListSrc.add(shadowedAddress);
                    break;
                }
            }
```

```
            for(String ipd: dstList) {
                if(shadowedAddress.equals(ipd)
                        || aputils.isIncludedIPString(shadowedAddress, ipd)) {
                    shadowedAddressesListDst.add(shadowedAddress);
                    break;
                }
            }
        }
        //Generate and add shadowing predicates
        for(String shadowed: shadowedAddressesListSrc) {
            if(!srcList.contains(shadowed)) {
                Predicate shpred = new Predicate(shadowed, false, "*", false, "*",
                        false, "*", false, L4ProtocolTypes.ANY);
                predicates.add(shpred);
            }
        }
        //Generate and add reconverted predicates
        for(String shadowed: shadowedAddressesListDst) {
            if(!dstList.contains(shadowed)) {
                Predicate rcvedpred = new Predicate("*", false, shadowed, false, "*",
                        false, "*", false, L4ProtocolTypes.ANY);
                predicates.add(rcvedpred);
            }
        }
        //Reconversion predicate
        if(!dstList.contains(node.getName())) {
            Predicate rcpred = new Predicate("*", false, node.getName(), false, "*",
                    false, "*", false, L4ProtocolTypes.ANY);
            predicates.add(rcpred);
        }
        //Add shadowed predicate: this is enough, all the others have already been added
        predicates.add(new Predicate(node.getName(), false, "*", false, "*",
                false, "*", false, L4ProtocolTypes.ANY));
    }
```

From "interesting" predicates to atomic predicates

The set of atomic predicates, based on the "interesting" predicates computed so far, is computed calling the function *computeAtomicPredicates*, described at the beginning of this Chapter.

```
atomicPredicates = aputils.computeAtomicPredicates(atomicPredicates, predicates);
```

A.2.3 Applying transformations

After the computation of the "interesting" predicates for the source and destination traffic of the requirements and for the forwarding behavior and transformation input domains of the nodes encountered along the path, and after having converted them into atomic predicates (line 7 of Algorithm 3), we can now apply the transformations. This operation (lines 8-15) consists of taking each predicate within the set of atomic predicates and compute how it is transformed after it has crossed each transformer.

In our case, which includes only NATs as transformers in the network, all the predicates resulting from the transformation are already present within the set of

already computed atomic predicates. So the if condition at line 11 is already true at the first iteration and the Algorithm stops immediately.

All that remains is to build the "transformation map" for each transformer, that is the function that describes the correspondence between the atomic predicate as it arrives in input and the relative atomic predicate (or predicates) eventually transformed and forwarded as it exits in output (e.g., "atomic predicate x crosses the NAT and is transformed into the atomic predicate y"). The implementation of this function is as follows.

```
for(Node node: transformersNode.values()) {
    HashMap<Integer, List<Integer>> resultMap =
            allocationNodes.get(node.getName()).getTransformationMap();
    if(node.getFunctionalType() == FunctionalTypes.NAT) {
        HashMap<String, List<Integer>> shadowingMap = new HashMap<>(); //grouped by dest address
        HashMap<String, List<Integer>> shadowedMap = new HashMap<>(); //grouped by dest address
        HashMap<String, List<Integer>> reconversionMap = new HashMap<>(); //grouped by source address
        HashMap<String, List<Integer>> reconvertedMap = new HashMap<>();  //grouped by source address
        List<Integer> notChaingingPredicateList = new ArrayList<>();
        List<IPAddress> natIPSrcAddressList = new ArrayList<>();
        for(String src: node.getConfiguration().getNat().getSource())
            natIPSrcAddressList.add(new IPAddress(src, false));
        IPAddress natIPAddress = new IPAddress(node.getName(), false);

        for(HashMap.Entry<Integer, Predicate> apEntry: networkAtomicPredicates.entrySet()) {
            Predicate ap = apEntry.getValue();
            //if source ip address list or dest ip address list have size != 1, it means it is a
            //complex predicates so it can not be a shodowing/reconversion predicates
            if(ap.getIPSrcListSize() != 1 || ap.getIPDstListSize() != 1) continue;
            if(ap.hasIPDstNotIncludedIn(natIPSrcAddressList)
                    && !ap.hasIPDstEqual(natIPAddress)) {
                if(ap.hasIPSrcEqual(natIPAddress)) {
                    //2*: if dest is not a src address of the NAT (so it is a public address)
                    //and ip source = ip NAT, this is a shadowed predicate
                    //{IP NAT, public address}
                    if(!shadowedMap.containsKey(ap.firstIPDstToString())) {
                        List<Integer> list = new ArrayList<>();
                        list.add(apEntry.getKey());
                        shadowedMap.put(ap.firstIPDstToString(), list);
                    } else {
                        shadowedMap.get(ap.firstIPDstToString()).add(apEntry.getKey());
                    }
                }
                else {
                    //1*: if dest is not a src address of the NAT (so it is a public address),
                    //while src is a src address of NAT (private address),
                    //this is a shadowing predicates {private address, public address}
                    if(ap.hasIPSrcEqualOrIncludedIn(natIPSrcAddressList))
                        if(!shadowingMap.containsKey(ap.firstIPDstToString())) {
                            List<Integer> list = new ArrayList<>();
                            list.add(apEntry.getKey());
                            shadowingMap.put(ap.firstIPDstToString(), list);
                        } else {
                            shadowingMap.get(ap.firstIPDstToString()).add(apEntry.getKey());
                        }
                }
            } else if(ap.hasIPSrcNotIncludedIn(natIPSrcAddressList)
                    && !ap.hasIPSrcEqual(natIPAddress)) {
                if(ap.hasIPDstEqual(natIPAddress)) {
                    //3*: src not included in NAT src, dest = IP NAT
                    //-> reconversion predicate {public address, IP NAT}
                    if(!reconversionMap.containsKey(ap.firstIPSrcToString())) {
                        List<Integer> list = new ArrayList<>();
                        list.add(apEntry.getKey());
```

```
                reconversionMap.put(ap.firstIPSrcToString(), list);
            } else {
                reconversionMap.get(ap.firstIPSrcToString()).add(apEntry.getKey());
            }
        } else if(ap.hasIPDstEqualOrIncludedIn(natIPSrcAddressList)) {
            //4*: src not included in NAT src, dest included in NAT src
            //-> reconverted predicate {public address, private address}
            if(!reconvertedMap.containsKey(ap.firstIPSrcToString())) {
                List<Integer> list = new ArrayList<>();
                list.add(apEntry.getKey());
                reconvertedMap.put(ap.firstIPSrcToString(), list);
            } else {
                reconvertedMap.get(ap.firstIPSrcToString()).add(apEntry.getKey());
            }
        }
    } else if(ap.hasIPSrcEqualOrIncludedIn(natIPSrcAddressList)
            && ap.hasIPDstEqualOrIncludedIn(natIPSrcAddressList)) {
        //5*: src included in NAT src (private) and dst included in NAT src (private)
        //-> predicate is just forwarded without transformation
        notChaingingPredicateList.add(apEntry.getKey());
    }
}
```

A.3 Algorithm 4: Atomic Flows computation

Let us now analyse the code that generates atomic flows, starting from the list of requirements and the set of computed atomic predicates, according to Algorithm 4. As mentioned in Chapter 6, this algorithm can be easily parallelized. In our case, we used a threadPool and a Runnable object to model the function as a task to be run by any thread inside the pool.

```
public void run() {
    Property prop = requirement.getOriginalProperty();
    String pSrc = prop.getSrcPort() != null &&
            !prop.getSrcPort().equals("null") ? prop.getSrcPort() : "*";
    //get all atomic predicates that match IPSrc and PSrc
    Predicate srcPredicate = new Predicate(prop.getSrc(), false, "*",
            false, pSrc, false, "*", false, L4ProtocolTypes.ANY);
    List<Integer> srcPredicateList = new ArrayList<>();
    for(HashMap.Entry<Integer, Predicate> apEntry: networkAtomicPredicates.entrySet()) {
        Predicate intersectionPredicate =
                aputils.computeIntersection(apEntry.getValue(), srcPredicate);
        if(intersectionPredicate != null
                && aputils.APCompare(intersectionPredicate, apEntry.getValue())
                && !apEntry.getValue().hasIPDstOnlyNegs()) {
            //System.out.print(apEntry.getKey() + " "); apEntry.getValue().print();
            srcPredicateList.add(apEntry.getKey());
        }
    }

    //System.out.println("Destination predicates");
    List<Integer> dstPredicateList = new ArrayList<>();
    String pDst = prop.getDstPort() != null &&
            !prop.getDstPort().equals("null") ? prop.getDstPort() : "*";
    Predicate dstPredicate = new Predicate("*", false, prop.getDst(), false, "*",
            false, pDst, false, prop.getLv4Proto());
    //get all atomic predicates that match IPDst and PDst and prototype
    for(HashMap.Entry<Integer, Predicate> apEntry: networkAtomicPredicates.entrySet()) {
        Predicate intersectionPredicate =
                aputils.computeIntersection(apEntry.getValue(), dstPredicate);
        if(intersectionPredicate != null
                && aputils.APCompare(intersectionPredicate, apEntry.getValue())) {
            //System.out.print(apEntry.getKey() + " "); apEntry.getValue().print();
```

```
            dstPredicateList.add(apEntry.getKey());
        }
    }

    //Generate atomic flows
    for(FlowPath flow: requirement.getFlowsMap().values()) {
        List<AllocationNode> path = flow.getPath();
        List<List<Integer>> resultList = new ArrayList<>();
        List<List<Integer>> resultListToDiscard = new ArrayList<>();
        //now we have the requirement, the path and the list of source predicates
        //-> call recursive function
        int nodeIndex = 0;
        for(Integer ap: srcPredicateList) {
            List<Integer> currentList = new ArrayList<>();
            recursiveGenerateAtomicPath(nodeIndex, requirement, path, ap, dstPredicateList,
                    resultList, resultListToDiscard, currentList);
        }

        for(List<Integer> atomicFlow: resultList) {
            flow.addAtomicFlow(atomicId.incrementAndGet(), atomicFlow);
        }
        for(List<Integer> atomicFlowToDiscard: resultListToDiscard)
            flow.addAtomicFlowToDiscard(atomicId.incrementAndGet(), atomicFlowToDiscard);
    }
}
```

This function is run once for each requirement. First the set B_0, described in the Algorithm as the set of all the atomic predicates matching with the input traffic of the requirement (IPSrc, pSrc), is computed. The same is done for the set of the atomic predicates matching with the destination traffic of the requirement (IPDst, pSrc, protoType). At the end, for each path of the requirement, the recursive *recursiveGenerateAtomicPath* function is called.

```
private void recursiveGenerateAtomicPath(int nodeIndex, SecurityRequirement sr,
        List<AllocationNode> path,
        int ap, List<Integer> dstPredicateList, List<List<Integer>> atomicFlowsList,
        List<List<Integer>> atomicFlowsListToDiscard, List<Integer> currentList) {
    AllocationNode currentNode = path.get(nodeIndex);
    Predicate currentPredicate = networkAtomicPredicates.get(ap);
    Predicate currentNodeDestPredicate = new Predicate("*", false,
            currentNode.getIpAddress(), false, "*", false, "*", false, L4ProtocolTypes.ANY);

    if(nodeIndex == path.size() -1) {
        //last node of the path
        if(dstPredicateList.contains(ap)) {
            //ALL OK, new atomic flow found
            atomicFlowsList.add(currentList);
            return;
        } else {
            //Discard path
            currentList.add(ap);
            atomicFlowsListToDiscard.add(currentList);
            return;
        }
    }

    Predicate intersectionPredicate =
            aputils.computeIntersection(currentPredicate, currentNodeDestPredicate);
    if(intersectionPredicate != null && aputils.APCompare(intersectionPredicate, currentPredicate)
            && (currentNode.getTransformationMap().isEmpty()  //not NAT
                    || (!currentNode.getTransformationMap().containsKey(ap)))) {
        //it is NAT but does not transform the predicate
        //Discard path: destination reached without reaching destination of the path
        currentList.add(ap);
```

```
            atomicFlowsListToDiscard.add(currentList);
            return;
        }

        //Apply transformation and filtering rules
        if(transformersNode.containsKey(currentNode.getIpAddress()) &&
                transformersNode.get(currentNode.getIpAddress())
                .getFunctionalType().equals(FunctionalTypes.NAT)) {
            if(currentNode.getTransformationMap().containsKey(ap)) {
                for(Integer newAp: currentNode.getTransformationMap().get(ap)) {
                    List<Integer> newCurrentList = new ArrayList<>(currentList);
                    newCurrentList.add(newAp);
                    recursiveGenerateAtomicPath(nodeIndex+1, sr, path, newAp,
                            dstPredicateList, atomicFlowsList, atomicFlowsListToDiscard, newCurrentList);
                }
            } else {
                //simple forwarding
                List<Integer> newCurrentList = new ArrayList<>(currentList);
                newCurrentList.add(ap);
                recursiveGenerateAtomicPath(nodeIndex+1, sr, path, ap,
                        dstPredicateList, atomicFlowsList, atomicFlowsListToDiscard, newCurrentList);
            }
        }
        else { //normal node
            List<Integer> newCurrentList = new ArrayList<>(currentList);
            newCurrentList.add(ap);
            recursiveGenerateAtomicPath(nodeIndex+1, sr, path, ap,
                    dstPredicateList, atomicFlowsList, atomicFlowsListToDiscard, newCurrentList);
        }
    }
```

In the first lines, the function checks if we have reached the end of the path. If we are in the destination node, then we check if the input predicate belongs to the destination traffic allowed for the requirement and computed above. In case it does, then the current atomic flow we are considering is added to the resulting set of atomic flows, *atomicFlowsList*. At the contrary, if it does not belong, then the current atomic flow is discarded and added to the set *atomicFlowsListToDiscard*.

At each step along the path, the destination IP of the incoming Predicate is put in intersection with the IP of the current node, to understand if the packet has reached its destination without reaching the destination of the requirement. In this latter case, the current atomic flows we are considering is discarded.

The incoming Predicate is also put in intersection with the transformation input domains of the node (in the case the node is a NAT). In case the intersection exists, then the Predicate is transformed into one or more new Predicates and the recursion continues in the following node. *recursiveGenerateAtomicPath* is called once for each new Predicate representing the result of the transformation.

In case the node is not a transformer, the Predicate is simply forwarded to the next node along the path without being affected by any transformation.

Appendix B

Implementation Maximal Flows

This Appendix shows a possible implementation in Java of Algorithm 5, described in Chapter 8. As for the implementation of the Atomic Flows, all the functions shown in this Appendix make use of the class Predicate described in Chapter 4.

B.1 Algorithm 5: Atomic Flows computation

The preliminary phase of the Algorithm, not shown in 5, consists of computing for each firewall the corresponding sets $\{I_a, I_d\}$, and for each transformer (NATs in our case) the corresponding transformation input domains. In the case of $\{I_a, I_d\}$, the procedure is exactly the same as the one done for Atomic Flows and also the code differs only slightly, so for the description we refer to the section A.2.2.

The procedure to compute the input domains for NATs, instead, changes. With this approach, transformations must no longer be modeled on the atomic predicates identifiers ("atomic predicate x is transformed into atomic predicate y") but on general predicates described by the class Predicate. It is therefore necessary to describe the various input domains for NATs as a disjunction of Predicates.

```
for(Node node: transformersNode.values()) {
    if(node.getFunctionalType() == FunctionalTypes.NAT) {
        List<IPAddress> sourceNatIPAddressList = new ArrayList<>();
        List<IPAddress> notSourceNatIPAddressList = new ArrayList<>();
        for(String ipSrc: node.getConfiguration().getNat().getSource()) {
            IPAddress natSrcAddress = new IPAddress(ipSrc, false);
            sourceNatIPAddressList.add(natSrcAddress);
            notSourceNatIPAddressList.add(new IPAddress(ipSrc, true));
        }
        notSourceNatIPAddressList.add(new IPAddress(node.getName(), true));

        //compute D1 transformation map
        List<Predicate> D1List = new ArrayList<>();
        for(IPAddress privateSrcAddress: sourceNatIPAddressList) {
            Predicate newPredicate = new Predicate("*", false, "*", false, "*",
                    false, "*", false, L4ProtocolTypes.ANY);
            List<IPAddress> srcIPList = new ArrayList<>();
            srcIPList.add(privateSrcAddress);
            newPredicate.setIPSrcList(srcIPList);
```

```
            newPredicate.setIPDstList(notSourceNatIPAddressList);
            D1List.add(newPredicate);
        }
        natD1map.put(node.getName(), D1List);

        //compute D2 transformation map
        Predicate D2Predicate = new Predicate("*", false, node.getName(), false, "*",
                false, "*", false, L4ProtocolTypes.ANY);
        D2Predicate.setIPSrcList(notSourceNatIPAddressList);
        natD2map.put(node.getName(), D2Predicate);

        //compute D31 transformation
        List<Predicate> D31List = new ArrayList<>();
        for(IPAddress privateSrcAddress1: sourceNatIPAddressList) {
            for(IPAddress privateSrcAddress2: sourceNatIPAddressList) {
                if(!privateSrcAddress1.equals(privateSrcAddress2)) {
                    List<IPAddress> srcList = new ArrayList<>();
                    List<IPAddress> dstList = new ArrayList<>();
                    srcList.add(privateSrcAddress1);
                    dstList.add(privateSrcAddress2);
                    Predicate newPredicate = new Predicate("*", false, "*", false, "*",
                            false, "*", false, L4ProtocolTypes.ANY);
                    newPredicate.setIPSrcList(srcList);
                    newPredicate.setIPDstList(dstList);
                    D31List.add(newPredicate);
                }
            }
        }
        natD31map.put(node.getName(), D31List);

        //Compute reconverted predicates
        List<Predicate> reconvertedList = new ArrayList<>();
        for(IPAddress privateSrcAddress: sourceNatIPAddressList) {
            List<IPAddress> dstList = new ArrayList<>();
            dstList.add(privateSrcAddress);
            Predicate newPredicate = new Predicate("*", false, "*", false, "*",
                    false, "*", false, L4ProtocolTypes.ANY);
            newPredicate.setIPSrcList(notSourceNatIPAddressList);
            newPredicate.setIPDstList(dstList);
            reconvertedList.add(newPredicate);
        }
        natReconvertedMap.put(node.getName(), reconvertedList);


        //Compute D32 transformation
        Predicate D32Predicate = new Predicate("*", false, "*", false, "*",
                false, "*", false, L4ProtocolTypes.ANY);
        D32Predicate.setIPSrcList(notSourceNatIPAddressList);
        D32Predicate.setIPDstList(notSourceNatIPAddressList);
        natD32map.put(node.getName(), D32Predicate);

}
```

B.1.1 Generate maximal flows

The function that generates the Maximal Flows starts by allocating a temporary list with Predicates of only wildcards, each one representing the class of all packets. Then the predicate for the source traffic of the requirement is generated and this will be forwarded along the considered path. As the Predicate progresses into the network, it is transformed and consequently the Maximal Flow list is updated.

```
private void generateMaximalFlows() {
    for(FlowPath flow : trafficFlowsMap.values()) {
        Property property = flow.getRequirement().getOriginalProperty();
        String pSrc = property.getSrcPort() != null && !property.getSrcPort().equals("null") ?
                property.getSrcPort() : "*";

        //Generate source predicate
        Predicate predicate = new Predicate(property.getSrc(), false, "*",
                false, pSrc, false, "*", false, L4ProtocolTypes.ANY);
        List<Predicate> currentMaximalFlow = new ArrayList<>();
        currentMaximalFlow.add(predicate);
        //preallocate the maximal flow list
        for(int i=1; i<flow.getPath().size(); i++) {
            Predicate voidPredicate = new Predicate("*", false, "*", false, "*",
                    false, "*", false, L4ProtocolTypes.ANY);
            currentMaximalFlow.add(voidPredicate);
        }

        if(flow.getPath().size() > 1) {
            recursiveGenerateMaximalFlowsForwardUpdate(1, flow.getRequirement(),
                    flow.getPath(), predicate, flow, currentMaximalFlow, false);
        }
    }
```

For each requirement and for each path belonging to the requirement, the function *recursiveGenerateMaximalFlowsForwardUpdate*, that starts the forward traversal, is called.

B.1.2 Forward traversal

```
private void recursiveGenerateMaximalFlowsForwardUpdate(int nodeIndex, SecurityRequirement sr,
        List<AllocationNode> path, Predicate inputPredicate,
        FlowPath currentFlowPath, List<Predicate> currentList,
        boolean somethingChanged) {

    if(nodeIndex >= path.size()) {
        return;
    }

    AllocationNode node = path.get(nodeIndex);

    if(nodeIndex == path.size() -1) {
        //We are in the last node of the path
        //Compute intersection with destination
        String dstPort = sr.getOriginalProperty().getDstPort() != null
                && !sr.getOriginalProperty().getDstPort().equals("null")
                ? sr.getOriginalProperty().getDstPort() : "*";
        L4ProtocolTypes proto = sr.getOriginalProperty().getLv4Proto() != null ?
                sr.getOriginalProperty().getLv4Proto() : L4ProtocolTypes.ANY;
        Predicate destPredicate = new Predicate("*", false, node.getIpAddress(), false,
                "*", false, dstPort, false, proto);
        Predicate intersectionPredicate = aputils.computeIntersection(destPredicate, inputPredicate);

        if(intersectionPredicate != null) {
            currentList.set(nodeIndex, intersectionPredicate);
            //start backward traversal
            recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path, intersectionPredicate,
                    currentFlowPath, currentList, false);
        }
        return;
    }
```

```
if(natD1map.containsKey(node.getIpAddress())) {
    //Node is a NAT
    //check if input Predicate has sourceIP == to nat IP
    List<IPAddress> natIPAddressList = new ArrayList<>();
    natIPAddressList.add(new IPAddress(node.getIpAddress(), false));
    if(aputils.APCompareIPAddressList(inputPredicate.getIPSrcList(), natIPAddressList)) {
        //the predicate has already been shadowed in previous traversals, so simply
        //change destination and forward
        Predicate newPredicate = new Predicate(currentList.get(nodeIndex));
        newPredicate.setIPDstList(inputPredicate.getIPDstList());
        newPredicate.setpDstList(inputPredicate.getpDstList());
        newPredicate.setProtoTypeList(inputPredicate.getProtoTypeList());
        currentList.set(nodeIndex, newPredicate);
        recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path, newPredicate,
                currentFlowPath, currentList, somethingChanged);
        return;
    }

    //check if it is a reconverted predicate. In that case forward the input packet
    //saved in this node changing only source
    boolean isReconverted = false;
    for(Predicate reconvertedPredicate: natReconvertedMap.get(node.getIpAddress())) {
        Predicate intersection =
                aputils.computeIntersection(inputPredicate, reconvertedPredicate);
        if(intersection != null && aputils.APCompare(intersection, inputPredicate)) {
            if(aputils.APCompareIPAddressList(currentList.get(nodeIndex).getIPDstList(),
                    natIPAddressList)) {
                isReconverted = true;
                break;
            }
        }
    }
    if(isReconverted) {
        Predicate newPredicate = new Predicate(currentList.get(nodeIndex));
        newPredicate.setIPSrcList(inputPredicate.getIPSrcList());
        newPredicate.setpDstList(inputPredicate.getpDstList());
        newPredicate.setProtoTypeList(inputPredicate.getProtoTypeList());
        currentList.set(nodeIndex, newPredicate);
        recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path, newPredicate,
                currentFlowPath, currentList, somethingChanged);
        return;
    }

    //Compute intersection with D1
    for(Predicate D1Predicate: natD1map.get(node.getIpAddress())) {
        Predicate intersectingD1Predicate =
                aputils.computeIntersection(D1Predicate, inputPredicate);
        if(intersectingD1Predicate != null) {
            //Do shadowing and generate a new flow
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            //change this node new input
            newCurrentList.set(nodeIndex, intersectingD1Predicate);
            //Generate new recursion with shadowed predicate as next
            //input predicate (to subsequent node)
            Predicate shadowedPredicate = new Predicate(intersectingD1Predicate);
            List<IPAddress> srcList = new ArrayList<>();


            shadowedPredicate.setIPSrcList(srcList);
            recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path, shadowedPredicate,
                    currentFlowPath, newCurrentList, true);
        }
    }
    //Compute intersection with D2
    Predicate intersectingD2Predicate =
            aputils.computeIntersection(natD2map.get(node.getIpAddress()), inputPredicate);
```

```
    if(intersectingD2Predicate != null) {
        //Do reconversion and generate new flows
        for(String natSrc: node.getNode().getConfiguration().getNat().getSource()) {
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            //change this node new input
            newCurrentList.set(nodeIndex, intersectingD2Predicate);
            //Generate new recursion with reconverted predicate as next
            //input predicate (to subsequent node)
            Predicate reconvertedPçredicate = new Predicate(intersectingD2Predicate);
            IPAddress natSrcAddress = new IPAddress(natSrc, false);
            List<IPAddress> dstList = new ArrayList<>();
            dstList.add(natSrcAddress);
            reconvertedPçredicate.setIPDstList(dstList);
            recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path,
                    reconvertedPçredicate, currentFlowPath, newCurrentList, true);
        }
    }
    //Compute intersection with D31
    for(Predicate D31Predicate: natD31map.get(node.getIpAddress())) {
        Predicate intersectingD31Predicate =
                aputils.computeIntersection(D31Predicate, inputPredicate);
        if(intersectingD31Predicate != null) {
            //change this node with new input
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            newCurrentList.set(nodeIndex, intersectingD31Predicate);
            //continue recursion without transformation
            recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path,
                    intersectingD31Predicate, currentFlowPath, newCurrentList, somethingChanged);
        }
    }
    //Compute intersection with D32
    Predicate intersectingD32Predicate =
            aputils.computeIntersection(natD32map.get(node.getIpAddress()), inputPredicate);
    if(intersectingD32Predicate != null) {
        //change this node with new input
        List<Predicate> newCurrentList = aputils.deepCopy(currentList);
        newCurrentList.set(nodeIndex, intersectingD32Predicate);
        //continue recursion without transformation
        recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path, intersectingD32Predicate,
                currentFlowPath, newCurrentList, somethingChanged);
    }
}
else if (allowedFirewallPredicates.containsKey(node.getIpAddress())) {
    //Check intersection with allowed and denied list
    List<Predicate> trasformedPredicates = new ArrayList<>();
    for(Predicate allowedPredicate: allowedFirewallPredicates.get(node.getIpAddress())) {
        Predicate intersectionAllowed =
                aputils.computeIntersection(allowedPredicate, inputPredicate);
        if(intersectionAllowed != null &&
                !aputils.APCompare(intersectionAllowed, inputPredicate))
            trasformedPredicates.add(intersectionAllowed);
    }
    for(Predicate deniedPredicate: deniedFirewallPredicates.get(node.getIpAddress())) {
        Predicate intersectionDenied =
                aputils.computeIntersection(deniedPredicate, inputPredicate);
        if(intersectionDenied != null &&
                !aputils.APCompare(intersectionDenied, inputPredicate))
            trasformedPredicates.add(intersectionDenied);
    }

    //Generate the new flows
    if(trasformedPredicates.size() > 0) {
        for(Predicate newPredicate:trasformedPredicates) {
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            newCurrentList.set(nodeIndex, newPredicate);
```

```
            //continue recursion without transformation
            recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path,
                    newPredicate, currentFlowPath, newCurrentList, somethingChanged);

        }
    } else {
        //simply forward the packet
        currentList.set(nodeIndex, new Predicate(inputPredicate));
        recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path, inputPredicate,
                currentFlowPath, currentList, somethingChanged);
    }
}
else {
    //node is a simple forwarder, just forward the predicate
    currentList.set(nodeIndex, new Predicate(inputPredicate));
    recursiveGenerateMaximalFlowsForwardUpdate(nodeIndex+1, sr, path, inputPredicate,
            currentFlowPath, currentList, somethingChanged);
}
```

First of all, the function checks if the predicate has reached the destination of the requirement (last node in the path). In case it does, then the predicate is put in intersection with the destination part of the requirement (IP destination, port destination, protocol type). If the intersection exists, then the backward traversal is called. Otherwise the current maximal flow is discarded.

If the node is not the last node of the path, it is an intermediate node, so we have to consider how this node can transform the predicate that arrives in input. There are mainly three considered cases for intermediate nodes: the node is a NAT, a firewall, or a simple allocation place (in this case the predicate is simply forwarded because the potential firewall allocated there is not yet configured).

In case the node is a **NAT**, the function must check if some intersection between the input predicate and the transformation domains of the NAT exists. For example:

- If the predicate has IP source equals to the IP of the NAT, this means that the predicate has already been shadowed by a previous forward traversal, so we have only to update the information related to the destination part of the predicate.
- If the predicate has already been reconverted by a previous forward/backward traversal, at the contrary, it is forwarded after updating its source part.
- If the predicate has intersection with D_1, the predicate is shadowed and recursion continues.
- If the predicate has intersection with D_2, then it is reconverted and new recursions starts (one for each possible reconverted predicate for the NAT).
- Finally, if the predicate has intersection with D_{31} and D_{32}, this means that no transformations are needed, so the predicate is simply forwarded.

In case the node is a **firewall**, the function must check if the intersection between the predicate and some filtering rules (expressed in Predicates as described above)

exists and if it is necessary to split the flow in different sub-flows. Then for each generated sub-flow, a new recursion starts.

Example:
the input predicate is {10.0.0.*, *, *, *} and there exists a filtering policy that blocks only {10.0.0.1, *, *, *}. In this case, the current considered flow is split into two different sub-flows, one with {10.0.0.1, *, *, *} as predicate in output, the other with {10.0.0.* ∧ !10.0.0.1, *, *, *}.

In case the node is a simple **forwarder**, the predicate is simply forwarded without modifications.

B.1.3 Backward traversal

The function that executes the backward traversal is very similar to the one related to the forward traversal. The only main difference is what happens when the predicate reaches the last node of the path (the first one in the forward traversal). In this case, the function must check if something has changed in the list of the current Maximal Flow for the previous forward and backward traversals. If no modifications have occurred, then there is no need for other forward and backward traversals and the algorithm can stop. Otherwise, if the boolean variable *somethingChanged* is set to true, this means that some modifications have occurred so it is necessary to start a new forward traversal which will be followed by a new backward traversal.

```
private void recursiveGenerateMaximalFlowsBackwardUpdate(int nodeIndex, SecurityRequirement sr,
        List<AllocationNode> path, Predicate inputPredicate,
        FlowPath currentFlowPath, List<Predicate> currentList, boolean somethingChanged) {

    if(nodeIndex < 0)
        return;

    AllocationNode node = path.get(nodeIndex);

    if(nodeIndex == 0) {
        //We are in the last first node of the path
        Predicate newPredicate = new Predicate(inputPredicate);
        currentList.set(nodeIndex, newPredicate);
        if(somethingChanged) {
            //start new forward update
            recursiveGenerateMaximalFlowsForwardUpdate(1, sr, path, inputPredicate,
                    currentFlowPath, currentList, false);
        } else {
            currentFlowPath.addMaximalFlow(maximalFlowId, currentList);
            maximalFlowId++;
        }
        return;
    }

    if(natD1map.containsKey(node.getIpAddress())) {
        //Node is a NAT
        //check if input Predicate has sourceIP == to nat IP
        List<IPAddress> natIPAddressList = new ArrayList<>();
        natIPAddressList.add(new IPAddress(node.getIpAddress(), false));
        if(aputils.APCompareIPAddressList(inputPredicate.getIPSrcList(), natIPAddressList)) {
            //the predicate has already been shadowed in previous traversals,
            //so simply change destination and forward
            Predicate newPredicate = new Predicate(currentList.get(nodeIndex));
```

```
        newPredicate.setIPDstList(inputPredicate.getIPDstList());
        newPredicate.setpDstList(inputPredicate.getpDstList());
        newPredicate.setProtoTypeList(inputPredicate.getProtoTypeList());
        currentList.set(nodeIndex, newPredicate);
        recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                newPredicate, currentFlowPath, currentList, somethingChanged);
        return;
    }

    //check if it is a reconverted predicate. In that case forward the input
    //packet saved in this node changing only source
    boolean isReconverted = false;
    for(Predicate reconvertedPredicate: natReconvertedMap.get(node.getIpAddress())) {
        if(aputils.computeIntersection(inputPredicate, reconvertedPredicate) != null) {
            if(aputils.APCompareIPAddressList(currentList.get(nodeIndex)
                    .getIPDstList(), natIPAddressList)) {
                isReconverted = true;
                break;
            }
        }
    }

    if(isReconverted) {
        Predicate newPredicate = new Predicate(currentList.get(nodeIndex));
        newPredicate.setIPSrcList(inputPredicate.getIPSrcList());
        newPredicate.setpDstList(inputPredicate.getpDstList());
        newPredicate.setProtoTypeList(inputPredicate.getProtoTypeList());
        currentList.set(nodeIndex, newPredicate);
        recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                newPredicate, currentFlowPath, currentList, somethingChanged);
        return;
    }

    //Compute intersection with D1
    for(Predicate D1Predicate: natD1map.get(node.getIpAddress())) {
        Predicate intersectingD1Predicate =
                aputils.computeIntersection(D1Predicate, inputPredicate);
        if(intersectingD1Predicate != null) {
            //Do shadowing and generate a new flow
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            //change this node new input
            newCurrentList.set(nodeIndex, intersectingD1Predicate);
            //Generate new recursion with shadowed predicate as next input predicate
            Predicate shadowedPredicate = new Predicate(intersectingD1Predicate);
            List<IPAddress> srcList = new ArrayList<>();
            srcList.add(new IPAddress(node.getIpAddress(), false));
            shadowedPredicate.setIPSrcList(srcList);
            recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                    shadowedPredicate, currentFlowPath, newCurrentList, true);
        }
    }
    //Compute intersection with D2
    Predicate intersectingD2Predicate =
            aputils.computeIntersection(natD2map.get(node.getIpAddress()), inputPredicate);
    if(intersectingD2Predicate != null) {
        //Do reconversion and generate new flows
        for(String natSrc: node.getNode().getConfiguration().getNat().getSource()) {
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            //change this node new input
            newCurrentList.set(nodeIndex, intersectingD2Predicate);
            //Generate new recursion with reconverted predicate as next input predicate
            Predicate reconvertedPçredicate = new Predicate(intersectingD2Predicate);
            IPAddress natSrcAddress = new IPAddress(natSrc, false);
            List<IPAddress> dstList = new ArrayList<>();
            dstList.add(natSrcAddress);
```

```
                recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                        reconvertedPçredicate, currentFlowPath, newCurrentList, true);
            }
        }
        //Compute intersection with D31
        for(Predicate D31Predicate: natD31map.get(node.getIpAddress())) {
            Predicate intersectingD31Predicate =
                    aputils.computeIntersection(D31Predicate, inputPredicate);
            if(intersectingD31Predicate != null) {
                //change this node with new input
                List<Predicate> newCurrentList = aputils.deepCopy(currentList);
                newCurrentList.set(nodeIndex, intersectingD31Predicate);
                //continue recursion without transformation
                recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                        intersectingD31Predicate, currentFlowPath, newCurrentList, somethingChanged);
            }
        }
        //Compute intersection with D32
        Predicate intersectingD32Predicate =
                aputils.computeIntersection(natD32map.get(node.getIpAddress()), inputPredicate);
        if(intersectingD32Predicate != null) {
            //change this node with new input
            List<Predicate> newCurrentList = aputils.deepCopy(currentList);
            newCurrentList.set(nodeIndex, intersectingD32Predicate);
            //continue recursion without transformation
            recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                    intersectingD32Predicate,
                    currentFlowPath, newCurrentList, somethingChanged);
        }
    }
    else if(allowedFirewallPredicates.containsKey(node.getIpAddress())) {
        //we are in a firewall
        //Check intersection with allowed and denied list
        List<Predicate> trasformedPredicates = new ArrayList<>();
        for(Predicate allowedPredicate: allowedFirewallPredicates.get(node.getIpAddress())) {
            Predicate intersectionAllowed =
                    aputils.computeIntersection(allowedPredicate, inputPredicate);
            if(intersectionAllowed != null &&
                    !aputils.APCompare(intersectionAllowed, inputPredicate))
                trasformedPredicates.add(intersectionAllowed);
        }
        for(Predicate deniedPredicate: deniedFirewallPredicates.get(node.getIpAddress())) {
            Predicate intersectionDenied =
                    aputils.computeIntersection(deniedPredicate, inputPredicate);
            if(intersectionDenied != null &&
                    !aputils.APCompare(intersectionDenied, inputPredicate))
                trasformedPredicates.add(intersectionDenied);
        }

        //Generate the new flows
        if(trasformedPredicates.size() > 0) {
            for(Predicate newPredicate:trasformedPredicates) {
                List<Predicate> newCurrentList = aputils.deepCopy(currentList);
                newCurrentList.set(nodeIndex, newPredicate);
                //continue recursion without transformation
                recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                        newPredicate, currentFlowPath, currentList, somethingChanged);
            }
        }
    }
    else {
        //Just forward the predicate
        Predicate newPredicate = new Predicate(inputPredicate);
        currentList.set(nodeIndex, newPredicate);
        recursiveGenerateMaximalFlowsBackwardUpdate(nodeIndex-1, sr, path,
                newPredicate, currentFlowPath, currentList, somethingChanged);
    }
}
```