Master of Science in Computer Engineering

Master Degree Thesis

Evaluation of a Heuristic Algorithm for Firewall Configuration

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

Candidate
David Di Marco

Academic Year 2023-2024

This work is subject to the Creative Commons Licence

Summary

In the ever-evolving landscape of network security and orchestration, the Network Functions Virtualization (NFV) paradigm is a novel innovative paradigm revolutionizing networking technology. NFV presents significant advantages over traditional networks by decoupling network functions from hardware appliances. This innovation enables the deployment of software processes as service functions on versatile general purpose servers. This approach introduces unparalleled flexibility and agility, enabling the creation of Service Graphs that generalize the Service Function Chain (SFC) concept. These graphs describe the organization and connection of network functions essential for end-to-end service delivery. However, a notable challenge arises in this context. While service designers are typically responsible for creating Service Graphs, security managers separately handle the allocation and configuration of Network Security Functions (NSFs) like firewalls. These manual operations are human error-prone and can result in latency when adapting to evolving security requirements.

To address these challenges, this thesis contributes to the advancement of the VEREFOO (VErified REFinement and Optimized Orchestration) framework, a Java-based system designed to automatically determine the optimal placement and configuration of network security mechanisms in a virtualized network, such as firewalls and anti-spam filters given as input the network topology and a set of Network Security Requirements (NSRs).

In prior studies, VEREFOO was explored through two distinct implementations. The initial approach is based on the formulation of the problem for optimization and verification and its execution using Z3, an advanced MaxSMT solving tool. This intricate problem-solving approach aims to meet a set of rigid constraints that are imperative for fulfillment. Simultaneously, it endeavors to attain the maximum cumulative value of specific weights assigned to soft clauses. The primary goal is to minimize the number of instances of NSFs for optimal resource utilization. Additionally, there is a focus on optimally configuring the rules in order to enhance the efficiency of filtering operations. Importantly, the formulation of the MaxSMT problem extends beyond optimization; it also serves the purpose of formal verification, ensuring the solution is formally correct using formal methods (correctness-by-construction). However, this solution faces scalability problems. Despite the optimal allocation of firewalls and rules, it still experiences significantly prolonged resolution times, yet when dealing with networks of small to medium sizes.

To tackle the scalability challenges inherent in the initial MaxSMT-based solution, an innovative heuristic algorithm has been introduced. In this context, a heuristic algorithm denotes an approach that, while not ensuring optimality, strikes

a balance between achieving suboptimal results and enhancing scalability. This strategic decision allows for the handling of considerably larger networks, although the resultant solution may not be optimal as the one proposed by the MaxSMT solver. It means that the solution given by the heuristic algorithm will result in a higher number of allocated NSFs and configured rules. Nevertheless, the noteworthy advantage lies in the capability to effectively manage vastly expanded network sizes, accompanied by substantially reduced resolution times when compared to the MaxSMT-based approach.

The main goal of this thesis is to compare these two distinct approaches to evaluate their performance in real-world scenarios and to improve the heuristic algorithm where possible. This assessment was conducted through a series of tests exploring feasibility, scalability, and optimality of both approaches. The aim is to determine the circumstances in which using one approach is more advantageous than the other, thus providing clear insights into optimal situations for the application of each method.

Acknowledgments

Alla mia famiglia.

Contents

List of Figures 10

List of Tables 12

Listings 13

1 Introduction 15

1.1 Thesis objective 15

1.2 Thesis description 17

2 Software Defined Networking and Network Function Virtualization 19

2.1 Limitations of traditional networks 19

2.2 Software-Defined Networks 20

2.2.1 Essential Concepts in Software-Defined Networking (SDN) . 20

2.2.2 Architectural Framework for Software Defined Networking (SDN) 21

2.3 Network Functions Virtualization 22

2.3.1 Essential Concepts in Network Functions Virtualization . . . 22

2.3.2 Architectural Framework for Network Functions Virtualizations (NFV) 23

3 VEREFOO 25

3.1 Foundation and Origins of VEREFOO 25

3.2 Service Graph 26

3.3 Allocation Graph 26

3.4 VEREFOO Architecture 28

3.5 Network Security Requirements 30

3.5.1 XML representation of the Network Security Requirements . 30

3.6 Traffic Flows Modeling 31

3.6.1 Predicates 32
3.6.2 Atomic Flows 33
3.6.3 Maximal Flows 34

4 The MaxSMT Problem 36
4.1 Maximum Satisfiability Modulo Theories 36
4.1.1 Boolean Satisfiability Problem (SAT) 36
4.1.2 Satisfiability Modulo Theories (SMT) 37
4.1.3 Maximum Satisfiability Modulo Theories (MaxSMT) 37
4.2 Z3 Theorem Prover 38
4.2.1 Z3 Architecture 39
4.2.2 Z3 Example 40

5 Heuristics in VEREFOO 41
5.1 The Heuristic Approach 41
5.2 Complete Heuristics 42
5.2.1 Initialization 43
5.2.2 Allocation 47
5.2.3 Configuration 48
5.3 Partial Heuristics 49
5.4 Alternative idea for Heuristics 50

6 Test Campaign Preparation: Code Interventions and New Topology Design 52
6.1 VEREFOO Implementations 52
6.2 Code Interventions 53
6.2.1 Allocation Places List Sorting 53
6.2.2 Correction of Heuristics-related Code 56
6.3 Topologies 57
6.4 Test Classes 63
6.5 Network Security Requirements Generator 64

7 Test Campaign - First Phase 66
7.1 First Scenario - Fixed Number of Endpoints and Variable Number of NSRs 68
7.1.1 100% Isolation Requirements 68

7.1.2 50% Isolation Requirements and 50% Reachability Requirements . . . 70
7.2 Second Scenario - Variable Number of Endpoints and Variable Number of NSRs . . . 73
7.2.1 100% Isolation Requirements . . . 73
7.2.2 50% Isolation Requirements and 50% Reachability Requirements . . . 75
7.3 Considerations . . . 77

8 Optimizing firewall Configuration for Minimal Rule Allocation 79
8.1 Eliminating Redundant Rule Configuration . . . 79
8.1.1 Elements class . . . 80
8.2 Post Processing of Rules . . . 81
8.3 Check with Verigraph . . . 82

9 Test Campaign - Second Phase 84
9.1 100% Isolation Requirements . . . 85
9.1.1 Execution Times . . . 85
9.1.2 Number of Allocated Firewalls . . . 87
9.1.3 Number of Configured Rules . . . 89
9.2 50% Isolation Requirements and 50% Reachability Requirements . . . 89
9.2.1 Execution Times . . . 89
9.2.2 Number of Allocated Firewalls . . . 92
9.2.3 Number of Configured Rules . . . 94
9.3 Evaluation of Differences Across Topologies . . . 96
9.3.1 MaxSMT Version - 100% Isolation Requirements . . . 96
9.3.2 MaxSMT Version - 50% Isolation Requirements and 50% Reachability Requirements . . . 99
9.3.3 Heuristics Version - 100% Isolation Requirements . . . 102
9.3.4 Heuristics Version - 50% Isolation Requirements and 50% Reachability Requirements . . . 105
9.4 Considerations . . . 108

10 Test Campaign - Third Phase 109
10.1 Optimality Tests . . . 110
10.1.1 Geant Topology . . . 110
10.1.2 Internet2 Topology . . . 112

10.1.3 Considerations 113
10.2 Time and Memory Limits in the Heuristic Approach 114
10.2.1 Geant Topology 114
10.2.2 Internet2 Topology 115
10.2.3 Considerations 115

11 Conclusions 116

Bibliography 118

List of Figures

2.1 Architecture of an SDN-based network 21

2.2 Architecture of the ETSI Framework for NFV 24

3.1 Overall Architecture of VEREFOO 28

4.1 Overall system architecture of Z3 39

5.1 Example of list L_A. The first row is the list of APs, with each of them being associated with a corresponding weight listed in the second row 46

5.2 Example of ordered list L_A 47

5.3 Allocation Place a_3 is extracted from list L_A 47

5.4 Visual example of the Branch-and-Bound method 51

6.1 Example of a simple Service Graph 54

6.2 Outcome of the framework using the wrong sorting method 55

6.3 Outcome of the framework using the right sorting method 56

6.4 VPNConfB topology 57

6.5 VPNConfB topology basic structures 58

6.6 Allocation Graph inspired by Geant network topology 59

6.7 Example of Geant topology adapted to VEREFOO 60

6.8 Allocation Graph inspired by Internet2 network topology 61

6.9 Example of Geant topology adapted to VEREFOO 62

7.1 Execution times 68

7.2 Number of allocated Firewalls 69

7.3 Number of configured rules 70

7.4 Execution times 71

7.5 Number of allocated Firewalls 72

7.6 Number of configured rules 72

7.7 Execution times 73

7.8 Number of allocated Firewalls 74
7.9 Number of configured rules 75
7.10 Execution times 75
7.11 Number of allocated Firewalls 76
7.12 Number of configured rules 77

9.1 Execution times comparison - GEANT 85
9.2 Execution times comparison - INTERNET2 86
9.3 Number of allocated firewalls - GEANT 87
9.4 Number of allocated firewalls - INTERNET2 88
9.5 Number of configured rules - GEANT and INTERNET2 89
9.6 Execution times comparison - GEANT 90
9.7 Execution times comparison - INTERNET2 91
9.8 Number of allocated firewalls - GEANT 92
9.9 Number of allocated firewalls - INTERNET2 93
9.10 Number of configured rules - GEANT 94
9.11 Number of configured rules - INTERNET2 95
9.12 Execution Times 96
9.13 Number of Allocated Firewalls 97
9.14 Number of Configured Rules 98
9.15 Execution Times 99
9.16 Number of Allocated Firewalls 100
9.17 Number of Configured Rules 101
9.18 Execution Times 102
9.19 Execution Times 103
9.20 Number of Configured Rules 104
9.21 Execution Times 105
9.22 Execution Times 106
9.23 Number of Configured Rules 107

10.1 110
10.2 111
10.3 111
10.4 112
10.5 112
10.6 113
10.7 Execution Time Limits 114
10.8 Execution Time Limits 115

List of Tables

6.1 Network Security Requirements of the example 54
6.2 Configuration of firewall 20.0.0.1 55
6.3 Configuration of firewall 20.0.0.2 55
6.4 Configuration of Firewall 20.0.0.4 56

Listings

3.1 XML representation of an Allocation Graph 31
4.1 Example of a MaxSMT problem expressed in Z3 language 40
4.2 Example of a SAT solution expressed in Z3 language 40
6.1 Sorting algorithm for Allocation Places list. 53
6.2 Fixed sorting order of Allocation Places' list. 54
8.1 Check whether a rule already exists on the firewall or not. 80
8.2 Check whether a rule already exists on the firewall or not. 80
8.3 Algorithm for post processing of rules. 81

Chapter 1

Introduction

1.1 Thesis objective

In the ever-evolving landscape of network technologies, two prominent innovations are reshaping the networking and security paradigms: Network Function Virtualization (NFV) and Software Defined Networking (SDN). These cutting-edge advancements represent transformative approaches to network architecture, with profound implications for enhancing the security landscape, amplified by the continual development of increasingly automated tools that can replace manual and error-prone operations.

NFV involves the abstraction of network functionalities from traditional hardware, enabling the creation of virtualized network instances on general-purpose hardware appliances. This dynamic approach facilitates efficient resource utilization and provides a flexible environment for deploying end to end connectivity, with strong security mechanisms. For example, a security task that is often both time-consuming and error-prone involves determining the placement and configuration of Network Security Functions (NSFs), like firewalls. These functions are essential for meeting specific Network Security Requirements (NSRs), which are essentially security constraints through which a specific behavior of the network is enforced. For instance, it could be required that a network node, that is compromised after an attack, must be isolated from a portion of the network in which there are servers that manage sensible informations. Alternatively, it may be necessary for a network node to have access to a specific network segment. In order to satisfy these isolation or reachability requirements, one or more NSFs must be properly placed and configured. Performing this task manually takes time, is prone to errors and may result in suboptimal outcomes. Automation, in this context, not only reduces the burden on human effort but also ensures provably correct and optimal solutions. The eventual formal correctness of the solution adds significant value, eliminating the need for extensive manual verification. Users can place high confidence in a solution generated through this automated approach. Optimal solutions contribute to resource efficiency by minimizing computational requirements and maximizing overall performance.

On the other hand, SDN introduces a paradigm shift by decoupling the control plane from the data plane. This separation allows for centralized control and

programmability, empowering administrators to dynamically configure and manage network behavior. The programmability inherent in SDN opens up new possibilities for implementing and adapting security measures in real-time.

The integration of these technologies holds high potential for fortifying network security. By abstracting network functions and centralizing control, security mechanisms can be dynamically deployed, configured, and adapted to emerging threats. This not only enhances the agility and responsiveness of security measures but also introduces a level of flexibility and scalability crucial for modern network environments.

The goal is to develop an automated tool capable of providing a dynamic and optimal allocation and configuration of NSFs on specific nodes within a network, based on the given logical topology and a set of NSRs. This tool aims to model the real-time behavior of the network, with a focus on minimizing response times to potential cyber attacks. While existing literature presents various approaches, none of them offers a fully automated solution to effectively mitigate the risk of human configuration errors. Some existing solutions are constrained to small-scale networks, encountering scalability issues. Additionally, certain approaches lack optimality and formal correctness.

VEREFOO (VErified REFinement and Optimized Orchestration) is a novel framework developed in Politecnico di Torino that manages the creation, configuration and orchestration of a complete end-to-end network security service. It was presented during the *2019 4th IEEE International Conference on Computing, Communications and Security (ICCCS)* ([1]). The initial version of the system was created with the aim of addressing and solving a Maximum Satisfiability Modulo Theories (MaxSMT) problem using Z3, a theorem solver developed by Microsoft Research, known for constructing solutions based on formal methods, ensuring correctness-by-construction. This solution consistently achieves optimality in terms of the number of firewalls allocated in the network and the configured rules. However, scalability concerns emerge when dealing with expansive networks, as the resolution time experiences exponential growth with the increasing number of endpoints in the network and the security requirements the framework must fulfill.

To address this challenge, a second version was developed, employing a heuristic algorithm. This alternative version excels in delivering faster resolutions, albeit offering a more approximate solution with respect to the solution based on the MaxSMT problem since it will result in a higher number of allocated NSFs and configured rules. It strategically navigates a trade-off between optimization, completeness, accuracy, and execution speed.

This thesis aims to contribute to the ongoing evolution of this innovative framework. The focus lies in empirically testing both approaches on real-world network topologies. The objective is to discern the situations where one approach proves more advantageous than the other, evaluating their performance across varying numbers of the network endpoints and security requirements. Multiple criteria will be considered to provide a comprehensive understanding of the framework's behavior in diverse scenarios, by means of scalability and performance tests.

1.2 Thesis description

Chapter 1 provides a concise overview of the thesis goals and its description. The subsequent sections are structured as follows:

- **Chapter** 2 provides a comprehensive exploration of SDN and NFV technologies. It offers insights into how these technologies operate, their functionalities, and the transformative changes they bring about.

- **Chapter** 3 provides an in-depth description of VEREFOO, a Java-based framework that this thesis aims to enhance. The focus of this chapter is to describe the operational principle of VEREFOO, delineating the input elements and output, with a more comprehensive exploration of the two methods employed for modeling traffic flows which are crucial for the tests that will be conducted during this thesis work: *Atomic Flows* and *Maximal Flows*.

- **Chapter** 4 aims to delve into the understanding of what is a MaxSMT problem, providing an overview of its origins and detailing the process of modeling NSRs in VEREFOO. Additionally, a section is dedicated to introducing the Z3 solver, a powerful tool employed to address the MaxSMT prolem.

- **Chapter** 5 begins by describing what a heuristic algorithm is. It then goes on to detail the heuristic algorithm implemented in VEREFOO, providing a thorough examination of each step: *initialization*, *allocation*, and *configuration*.

- **Chapter** 6 delineates corrective and preemptive actions taken prior to the beginning of test campaigns. Specifically, it shows code corrections, tools used, and topologies employed during testing.

- **Chapter** 7 broadly illustrates the differences between the MaxSMT-based version and the heuristic-based version, as well as between the model using Atomic Predicates and the one utilizing Maximal Flows. Its focus is on scalability, execution times and optimality.

- **Chapter** 8 elucidates the enhancements made to the heuristic algorithm. In particular, given that the first test campaign revealed the heuristic allocating significantly more rules than the solver for the same topology, the chapter presents improvements made to optimize the number of rules configured by the heuristic. Additionally, it demonstrates the validation process using Verigraph.

- **Chapter** 9 presents the results obtained from a second test campaign. This time, the tests were conducted on two new topologies, primarily aimed at showing the effectiveness of the new rule post-processing algorithm. Additionally, it highlights differences in the framework's behavior across different topologies.

- **Chapter** 10 presents the results of the third and final test campaign. This time, the primary focus shifts to optimality rather than scalability. Additionally, it shows the maximum time limits that the heuristic algorithm can reach before exhausting the test IDE's memory.
- **Chapter** 11 concludes this thesis work by summarizing the accomplished achievements and outlining potential future works.

Chapter 2

Software Defined Networking and Network Function Virtualization

This chapter delves into the transformative impact of Software Defined Networking (SDN) and Network Function Virtualization (NFV) on modern network architecture and operations. SDN revolutionizes network management by separating the control plane from the data plane, allowing for centralized control, programmability, and dynamic configuration. NFV complements this by virtualizing network functions, liberating them from hardware constraints and optimizing resource utilization, scalability, and security.

The chapter explores the core principles of SDN, highlighting its benefits in efficient resource management and orchestration. Similarly, NFV's role in transforming network functions into software-based entities is elucidated, emphasizing its contributions to scalability, security, and rapid deployment of network services.

Furthermore, the chapter discusses how these technologies can be leveraged to automate security aspects within networks, showcasing their combined potential in overcoming limitations of traditional networking paradigms.

2.1 Limitations of traditional networks

Traditional networks face various limitations in terms of flexibility, scalability, and efficient resource usage which are resolved by these new two technologies. Some of them are listed below:

- **Rigidity and Configuration Complexity:** In traditional networks, configuration and resource management can be complex and inflexible. SDN separates traffic control from routing table management, enabling dynamic and flexible network configuration.
- **Limited Scalability:** Traditional networks may struggle to efficiently handle sudden increases in resource demand. SDN allows for centralized and programmable network resource management, facilitating scalability through automated network operations.

- **High Costs and Inefficient Resource Usage:** Traditional networks often require dedicated hardware for specific network functions. NFV consolidates network functions into virtual machines running on standard servers, reducing hardware costs and improving resource usage.

- **Challenges in Security Policy Management:** Traditional networks may struggle to apply security policies flexibly and dynamically. SDN streamlines security policy management by providing centralized control over network traffic through a central controller.

- **Low Automation:** Traditional networks often rely on manual intervention for configuration, monitoring, and optimization. SDN and NFV introduce higher levels of automation, allowing networks to dynamically adapt to changing needs and traffic conditions without intensive manual management.

2.2 Software-Defined Networks

2.2.1 Essential Concepts in Software-Defined Networking (SDN)

Software-Defined Networking (SDN), as described in [2], is a networking paradigm that uses software-based controllers or APIs to communicate with underlying hardware infrastructure and control traffic flow across a network. Unlike traditional architectures relying on dedicated hardware devices like routers and switches, SDN enables virtual network creation and management, as well as control over traditional hardware, all through software-based mechanisms. This approach introduces a centralized server for routing data packets, marking a significant shift in network traffic orchestration and management compared to traditional methods.

It id based on 4 main concepts:

- **Data Plane:** this layer manages the dynamic transmission and reception of network traffic, including both physical infrastructure and virtual components. Rules and forwarding tables guide the effective packet forwarding to the desired destination.

- **Control Plane:** it makes decisions regarding the optimal route for network traffic. It includes complex functions such as routing algorithms, switching mechanisms, and network management strategies. The control plane communicates constantly with the data plane, providing precise instructions on how to manage, route, and prioritize traffic.

- **Separation of Data Plane and Control Plane:** SDN introduces a separation between the data plane and the control plane, allowing a centralized controller to manage the network strategy without directly handling data forwarding.

- **APIs for Communication:** Northbound and Southbound interfaces are the connection points between the various layers of SDN. They enable communication between the control plane and the data plane, as well as between the control plane and external applications. These APIs allow the controller to communicate programmatically with network devices, providing instructions on how to manage traffic, update forwarding tables, and adapt to evolving network conditions.

2.2.2 Architectural Framework for Software Defined Networking (SDN)

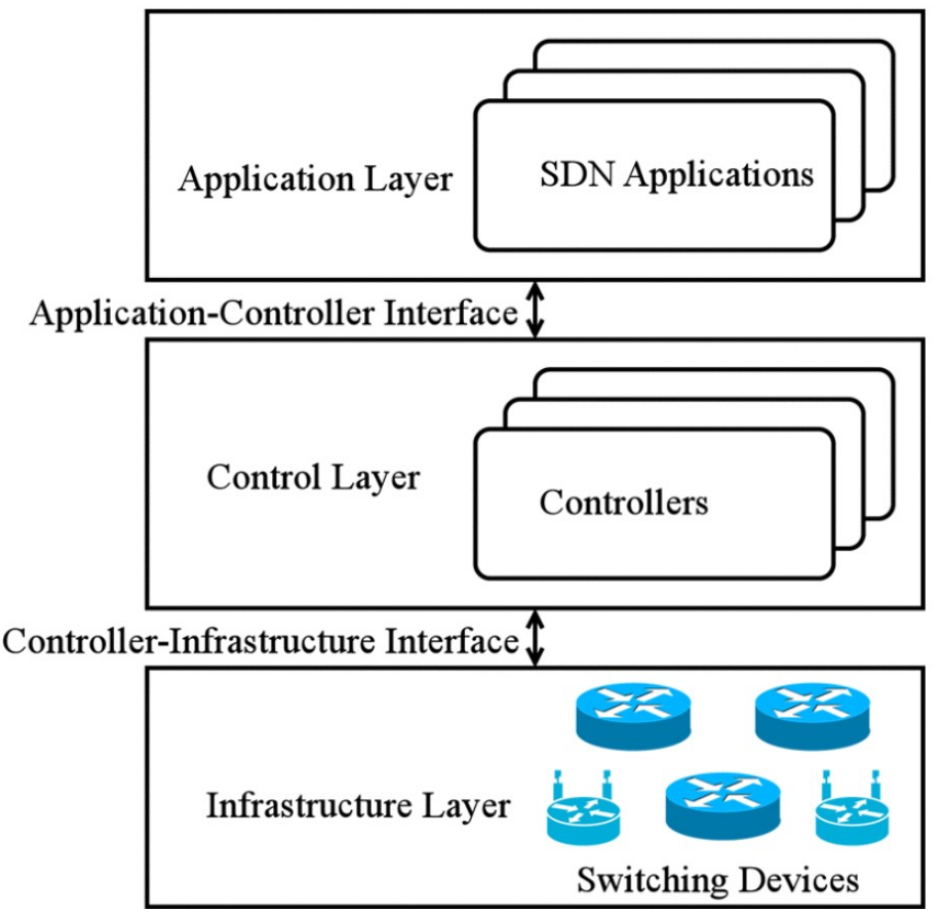


Figure 2.1: Architecture of an SDN-based network

The Open Networking Foundation (ONF) [3] is a non-profit consortium dedicated to the development, standardization, and commercialization of SDN technology. It has proposed a reference model for SDN, as illustrated in Figure 2.1, consisting of three distinct layers which are described below.

- **Infrastructure Layer**: it includes switching devices like switches and routers in the data plane. These devices collect network status and process packets based on rules from a central controller. SDN allows these devices to be generic and efficient, unlike traditional vendor-specific hardware. Each switch

has forwarding tables with rules and actions for packet handling, eliminating complex configurations and promoting interoperability.

- **Control Layer**: it acts as a bridge between the application and infrastructure layers, providing interfaces for controllers to access network capabilities and for SDN applications to access network status and configure packet forwarding rules. It also enables communication among controllers for network-wide coordination.

- **Application Layer**: it hosts SDN applications tailored to user needs, leveraging the programmable platform provided by the control layer. Examples of SDN applications include dynamic access control, seamless mobility, server load balancing, and network virtualization.

2.3 Network Functions Virtualization

2.3.1 Essential Concepts in Network Functions Virtualization

The concept of Network Function Virtualization (NFV) constitutes a natural progression in the evolution of networking, representing a seamless extension of the SDN paradigm described in Section 2.2. While SDN focuses on the centralization and programmability of network control, NFV takes a step further by addressing the virtualization of network functions that were traditionally implemented on dedicated hardware. Without NFV, the inherent limitations of the SDN paradigm would be pronounced, as the hardware devices it interacts with lack dynamism and adaptability. These devices are typically engineered for static execution of specific functions. However, NFV introduces a transformative dimension by virtualizing network functions that were traditionally bound to hardware, effectively turning them into versatile software entities.

Through the synergy of NFV and SDN, these virtualized functions can seamlessly be deployed on generic hardware devices. This symbiotic relationship empowers a hardware device to dynamically alter its behavior in response to evolving requirements, a capability unattainable in a static hardware-centric environment. This dynamic adaptability is crucial in addressing the evolving needs and challenges of modern networks, ensuring a more agile and responsive network infrastructure.

NFV is based on 4 principles:

- **Virtualization:** NFV uses advanced virtualization technologies to create virtual versions of network functions. This innovative method frees software from its hardware constraints, bringing about increased flexibility and resource efficiency in network architectures. This transformative approach marks a significant shift in how networks operate, allowing for more adaptability and efficient use of resources.

- **Abstraction:** at the heart of NFV is the concept of abstraction, where network functions are separated from the underlying hardware. This strategic abstraction creates a clear divide, making it easier to manage and orchestrate network services. It acts as a channel for smooth adaptability and smart use of resources, simplifying the way networks are operated.

- **Automation:** NFV relies on automation to streamline tasks such as deployment, configuration, and management of network functions. This automated approach contributes to operational efficiency by reducing manual intervention, enabling quick and precise scaling, and optimizing resource allocation. Automation in NFV ensures dynamic scaling, rapid deployment, and adaptability to changing conditions, contributing to the overall effectiveness and benefits of NFV in modern networking environments.

- **Standardization:** an essential principle in NFV is its strong focus on standardization, that is led by ETSI (European Telecommunications Standards Institute) [4]. By promoting the use of standardized interfaces and protocols, NFV creates an environment where different vendors and components can seamlessly work together. This commitment to standardization ensures that various vendors' technologies can easily integrate and collaborate, making the overall system more compatible and straightforward. It simplifies the incorporation of a diverse range of network functions into the NFV framework.

2.3.2 Architectural Framework for Network Functions Virtualizations (NFV)

To implement these management roles and keep the system open and non-proprietary, a framework must be defined for standardization. This standard framework should ensure that the VNF deployed is not tied to specific hardware and does not need to be especially tailored for any environment. It should offer vendors a reference architecture that they can follow for consistency and uniformity in the deployment methodologies of any VNF they implement.

ETSI proposed the framework illustrated in Figure 2.2.

The process of Function Virtualization is divided into three high-level blocks:

- **Network Functions Virtualization Infrastructure (NFVI):** This block includes three main components: physical hardware resources divided into computing, storage, and network categories; the virtualization layer, which directly interacts with the hardware pool, making it accessible to VNFs as virtual machines; and virtual resources that are made available to the upper layer.

- **Virtualized Network Functions (VNFs):** The VNF layer is where the virtualization of network functions takes place, consisting of the VNF-block and its managing entity, the VNF-Manager (VNFM). The VNF-block is a combination of VNF and Element Management (EM) blocks. In this layer, a virtualized network function is designed to run on hardware with sufficient

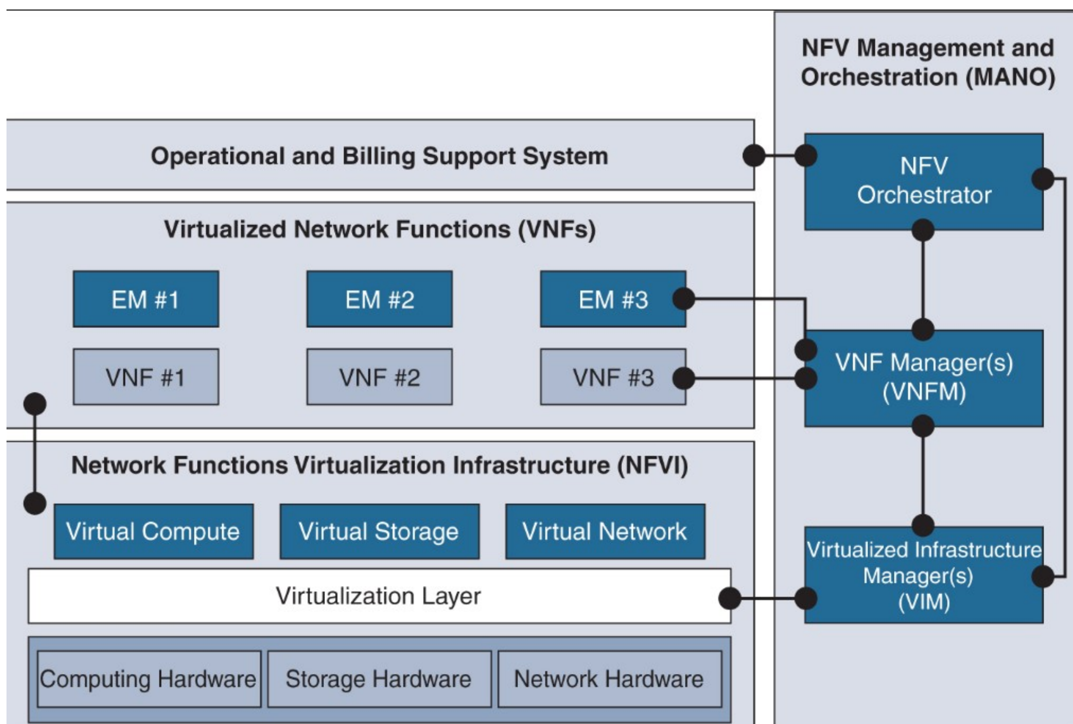


Figure 2.2: Architecture of the ETSI Framework for NFV

computing, storage, and network interfaces. However, the VNF is designed to be oblivious to the details of the virtualized environment, running on generic hardware that is essentially a virtual machine. The VNF is expected to behave and present external interfaces identical to the physical implementation of the network function and device it is virtualizing.

- **Management and Orchestration (MANO):** the MANO layer is established as an independent block in the architecture, which interacts with both the NFVI and VNF blocks. Within the framework, MANO is responsible for managing all resources in the infrastructure layer. This involves tasks like creating and deleting resources, as well as handling the allocation of resources for the VNFs.

Chapter 3

VEREFOO

3.1 Foundation and Origins of VEREFOO

The increasing prominence of cybersecurity attacks underscores the criticality of addressing misconfigurations in Network Security Functions like firewalls and VPN terminators. Manual approaches by network administrators often result suboptimal distribution of filtering or protection rules on NSFs, leading to inherent vulnerabilities.

In response to this challenge, there is a growing imperative to introduce automated policy-based network security management tools. These tools aim to support human operators in creating and configuring security services through an automatic process. This process is responsible for establishing policies for each NSF, ensuring adherence to specified security requirements or intents. The advantages of Security Automation are evident, including the prevention of human errors, automatic conflict analysis of policies, and formal verification of their correctness.

The advent of cutting-edge technologies such as SDN and NFV has brought automation into a pivotal role within the realm of cybersecurity. Leveraging these technologies allows for the comprehensive harnessing of automation, facilitating the implementation of security mechanisms that are not only more robust but also highly effective. Despite these promising prospects, the research in this domain is still in its early stages. Nowdays, diverse approaches in leveraging these technological advancements to enhance cybersecurity have been published in literature. For example, [5] and [6] establish policy-based automated methodologies for configuring firewalls to meet specific security requirements. However, these methodologies often fall short in providing formal assurance of correctness. Moreover, their design is tailored for traditional networks utilizing hardware appliances, neglecting adaptation to the dynamic environments of NFV. On the contrary, alternative works, such as the one highlighted in [7], leverage automation and formal verification to fix firewall misconfigurations. However, these approaches do not facilitate the creation of firewall policies from scratch. Another notable work, as discussed in [8], has an approach similar to the one developed in VEREFOO but does not focus on a larger pool of network functions among which to choose for enforcing the security policies.

Before delving into the workings of VEREFOO, it is useful to understand the

concepts of *Service Graph* (SG) and *Allocation Graph* (AG), as well as their differences. The first is showed in Section 3.2, while the second is explained in Section 3.3.

3.2 Service Graph

A **Service Graph** (SG) is a logical topology in networking that consists of a set of interconnected Network Functions (NFs) including examples likes load balancing, web-caching, NATs and forwarders. They are used by a service designer responsible for defining a complete end-to-end network service. It can be considered as a generalization of the Service Function Chain (SFC), an ordered set of abstract service functions and constraints that must be applied on selected packets/flows. The difference is that in the Service Graph, the functions do not need to be placed in a linear combination, but they can be organized in a more complex graph structure that allows for multiple paths between endpoints and can include loops.

In VEREFOO, we can represent the Service Graph as

$$G_S = (N_S, L_S)$$

where:

1. N_S is the set of all the nodes of the Service Graph like the end points and the NFs. This expression can be represented as

   $$N_S = E_S \cup S_S$$

   where E_S denotes the collection of endpoints directly associated with a terminal or subnetwork in the substrate infrastructure where virtual instances of functions are physically deployed. Conversely, S_S represents the set of service functions; these elements in S_S are basic NFs that do not provide security protection against cyberattacks but are solely utilized to establish an end-to-end service.

2. L_S is the set of edges representing the directed connections between the nodes, i.e between a pair of elements of N_S.

In this way, the SG provides an abstract view of the network including all the possible paths a packet could follow. One thing to note is that in the SG, security functions like firewalls are not involved, but the primary goal is to provide a complete network service to the user.

3.3 Allocation Graph

An **Allocation Graph** (AG) is a logical topology which can either be generated from scratch or derived from a SG. It shares the same set of NFs as the SG but incorporates additional nodes known as Allocation Places (APs), which are placed

on each link that connects two consecutive nodes. These places act as placeholders where a NSF, like a firewall, might be allocated. The configuration of NSF in the AG is automatically provided. If the decision is made not to allocate a NSF within an AP, but the AP is part of the path for at least one input requirement, its position will be occupied by a forwarder. This is because the forwarder's function would be to forward each received packet along the designated path.

Formally, the Allocation Graph is a directed graph that can be represented as

$$G_A = (N_A, L_A)$$

where:

1. N_A is the set of all the network nodes of the Allocation Graph.
2. L_A is the set of edges representing the directed connections between the nodes, i.e between a pair of elements of N_A.

The difference with the SG lies in the definition of the set of nodes

$$N_A$$

. Indeed, in an Allocation Graph, this set is modeled as

$$N_A = E_A \cup S_A \cup P_A$$

where:

1. E_A is the set of end points of the topology. We can say that

$$E_A = E_S$$

2. S_A is the set of service functions. Even in this case, we can say that

$$S_A = S_S$$

3. P_A is the set of the Allocation Places that have been instanciated by the transformation process from the SG to the AG.

One important thing to say is that, although the transformation process from the SG to the AG is automatic, the service designer has the authority to impose constraints on the generation process, either forcing the allocation of a NSF to a specific AP or forbidding the placement of a new AP in a designated location. This feature increases flexibility and decreases the computation time, since the space of solutions is more restricted, but it can lead to unoptimized solutions.

3.4 VEREFOO Architecture

As reported in [1], "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".

Figure 3.1 illustrates the overall architecture of VEREFOO, followed by a description of its modules.

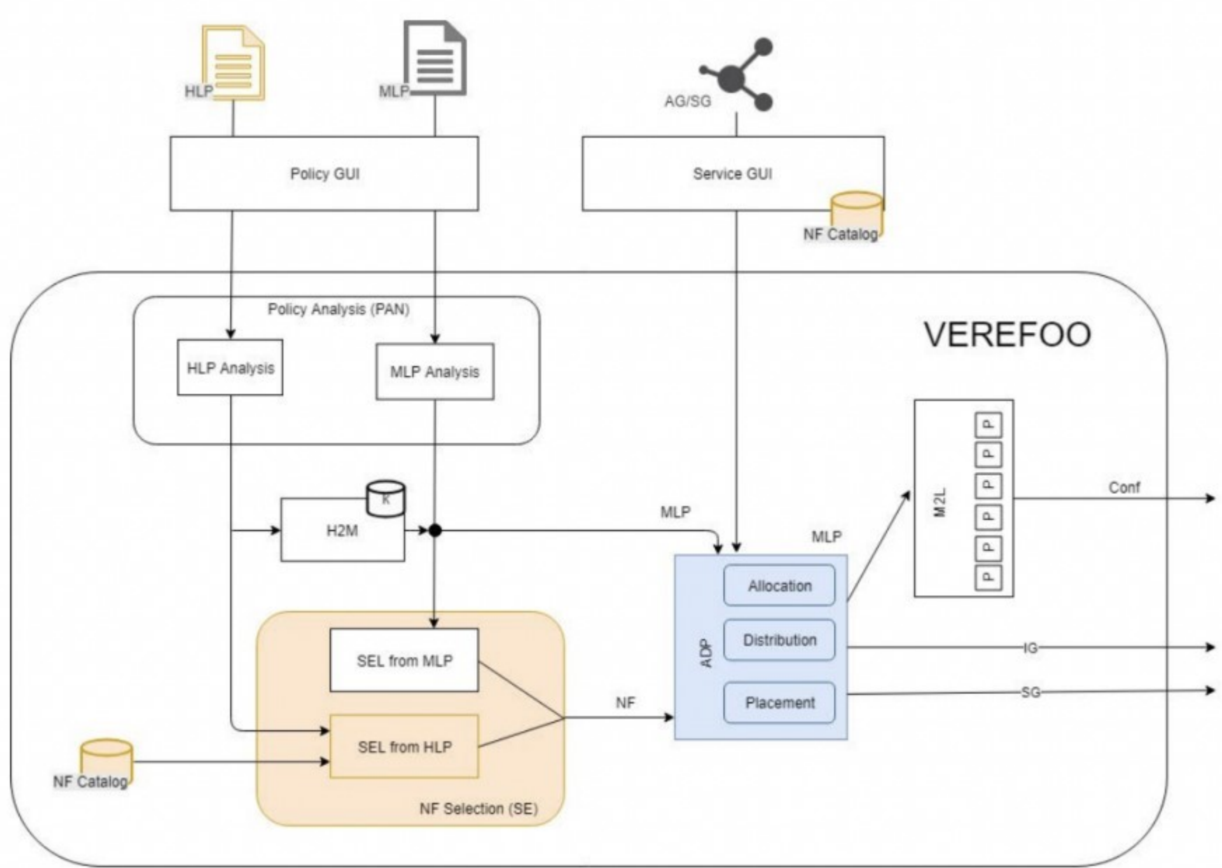


Figure 3.1: Overall Architecture of VEREFOO

- The starting point is the set of NSRs that the framework must fulfill. These requirements are provided by the user through the **Policy GUI** and can be expressed in two different ways: HLP or MLP. The choice between these formats depends on the user's level of expertise. HLP refers to *High-Level Policies*, articulated in a user-friendly and easily understandable language. On the other hand, MLP stands for *Medium-Level Policies*, which are lower-level requirements expressed in terms of ports and IP addresses, offering more specificity and detail. The flexibility in choosing between these formats allows users to adopt an approach that best aligns with their skills and needs.

- If one chooses to express certain NSRs using the HLP language, these requirements are then translated into the MLP language by the **High-To-Medium Module (H2M)**. The MLP language encompasses all the necessary details for the framework to subsequently create the corresponding network security functions that will satisfy them.

- Once all requirements are expressed in the same language (MLP), they are input into the **Policy Analysis (PAN)** module. It verifies the consistency and absence of conflicts among the requirements, while also identifying any errors. Upon successful verification, the module will return the minimum set of requirements that must be satisfied. However, in cases where errors cannot be resolved automatically, a detailed report describing them is generated.

- The next step is carried out by the **NF Selection Module (SE)**, whose task is to read the requirements and choose the most suitable NSFs to fulfill them. They are selected from a pre-built catalogue (the **NF catalogue**) containing all the functions that the framework can manage.

- At this point, the framework needs to determine where to allocate the NSFs. To accomplish this, the user must provide a second input to the framework through the **Service GUI**. Through this module, the user can define the Service Graph (or directly the Allocation Graph) which outlines the network topology that needs to include security functions to meet the previously expressed requirements. This module is also linked to the NF catalog, enabling the user to select functions to be included in the graph.

- The core of the framework is the **Allocation, Distribution and Placement Module (ADP)**. It takes as input the medium-level NSRs, the list of selected NSFs, and the original Service Graph. Its primary goal is to compute the final Service Graph, incorporating the additional NSFs. To achieve this, the ADP module uses a partial weighted MaxSMT problem solver, z3Opt (presented in Chapter 4). The security requirements are introduced into the solver as non-relaxable constraints that must be satisfied at all costs. Simultaneously, other specifications can be introduced as weighted and optional soft constraints to find the ideal and optimal allocation of NSFs within the network. While the MaxSMT problem is NP-complete in terms of computational complexity, a well-formulated problem and appropriate pruning techniques can potentially reduce the complexity to polynomial time. In the next section of this chapter, we will delve into a detailed discussion of the various types of constraints applied in VEREFOO.

- At the end, the last module is the **Medium-to-Low Module (M2L)** which takes a list of medium-level policy rules generated by the solver as input and converts them into low-level language, which depends on the practical implementation of the network functions..

3.5 Network Security Requirements

In Section 3.4, we mentioned that one of the two inputs to be provided to the framework consists of a set of Network Security Requirements that the framework must fulfill by incorporating Network Security Functions in the Allocation Graph. While various types of NSRs can be expressed, this thesis specifically focuses on *connectivity requirements*, which manifest in two distinct types:

1. **reachability property**, when an end point must be able to reach another one if at least one path between them exists;
2. **isolation property**, when an end point must not be able to reach another one through all possible paths interconnecting them.

Moreover, VEREFOO provides the service designer with the option to specify one of three potential *general behaviors*, described below. Each behavior is defined by a *default approach*, outlining how the framework should handle all traffic flows that lack explicitly formulated requirements.

1. **Whitelisting**, if all the communications for which no specific requirements have been formulated should be blocked. In this case, by default all traffic is blocked and the user only specifies reachability requirements;
2. **Blacklisting**, if all the communications for which no specific requirements have been formulated should be allowed. In this case, by default all traffic is allowed and the user only specifies isolation requirements;
3. **Specific**, if the service designer doesn't care about the communications for which no specific requirements have been formulated. In this case, the user can specify both reachability and isolation requirements and the framework autonomously decides how to handle the other traffic flows.

It's important to note that when opting for the third approach, it is imperative for the requirements to be conflict-free. This aspect is addressed by the PAN module in VEREFOO. In contrast, the other two approaches don't need conflict resolution, as all requirements would be either reachability or isolation, depending on the chosen approach. In this thesis, we adopt the specific approach, assuming that the service designer explicitly specifies which communications should be blocked or allowed. Additionally, a second assumption is made that the input set of NSRs is consistently conflict-free.

3.5.1 XML representation of the Network Security Requirements

The XML schema for the NSRs is defined through the *PropertyDefinition* element. It comprises a list of *Property* elements, each corresponding to a single requirement. Each requirement is identified by a unique identifier associated with the corresponding Service Graph or Allocation Graph, along with some attributes which

are described below. Listing 3.1 provides an example of a reachability requirement and an isolation requirement.

```
<PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty" src="10.0.0.-1"
dst="20.0.0.3" dst_port="60" />
    <Property graph="0" name="IsolationProperty" src="20.0.0.2"
dst="30.0.0.1" src_port="80" dst_port="[100-200]" />
</PropertyDefinition>
```

Listing 3.1: XML representation of an Allocation Graph

Each *Property* element is characterized by the following attributes:

- **name** is the kind of requirement, in our case it can be *ReachabilityProperty* or *IsolationProperty*;
- **src** is the IP address of the source node of the requirement;
- **dst** is the IP address of the destination node of the requirement;
- **src_port** is the number or interval of numbers of the source port;
- **dst_port** is the number or interval of numbers of the destination port;
- **l4_proto** is the type of layer 4 protocol (for example TCP or UDP).

It's important to note that the source and destination ports are optional fields; if left unspecified, the entire range of ports will be considered. In Listing 3.1, two requirements are illustrated, each representing a different type. The first requirement denotes a reachability condition for communications originating from the subnet 10.0.0.0/24 (expressed using a wildcard, denoted by the value -1) to the IP address 20.0.0.3 on port 60. Conversely, the second requirement is an isolation condition for communications originating from node 20.0.0.2 and port 80, destined for IP address 30.0.0.1, listening on a port range between 100 and 200.

3.6 Traffic Flows Modeling

To enable VEREFOO to achieve its goal, which is to satisfy the provided NSRs, rquires selecting a model that is both efficient and high-performing for network traffic. Specifically, this pertains to the traffic exchanged among nodes, which serves as the foundational element for modeling NSRs. It is imperative to have a formally correct representation that is simultaneously computationally efficient for subsequent operations performed by the solver or by the heuristics.

Essentially, the primary objective is to establish a formal model that accurately represents the journey of packets originating from a communication source. This includes modeling how these packets are forwarded or modified within a network until they reach their destination. This objective can be broken down into distinct sub-goals, encompassing the modeling of packet traversal, potential paths within

a network, and the transformations executed by network functions on packets in transit.

In this section, the focus is on introducing two optimized traffic flow models, namely Atomic Flows and Maximal Flows, which were initially introduced in VEREFOO by a previous work [9]. The significance of presenting these models within the context of this thesis lies in their role in highlighting the differences between them. In fact, one of the objectives of this thesis is to examine and compare these two models to discern their relative advantages and disadvantages both in the solver and in the heuristic algorithm.

The traffic flow models play a crucial role in formal and automated security management processes by representing, identifying, and consolidating categories of network packets, commonly referred to as traffics. These models are used to compute the routing and transformation of packets as they navigate through different network nodes. This knowledge is pivotal in determining the most effective placement and setup of NSFs in alignment with user-specified NSRs.

3.6.1 Predicates

A packet is conceptualized within the model as a predicate derived from certain fields, particularly segments of its header. Packets sharing identical characteristics in these specified fields are classified into the same category and are denoted by a common predicate. As a result, all nodes encountered in the network treat packets within the same predicate in a uniform manner. Note that a predicate essentially signifies what is referred to as network traffic. The decisions regarding packet forwarding domains and transformation actions are exclusively determined by the predicate to which the packet belongs.

Within the network model under consideration, each node possesses specific properties that define the element allocated at that position.

- **Access Control List (ACL)**: A node contains a collection of input and output ports, each one governed by an ACL dictating whether a packet with specific header attributes can traverse the port. This mechanism establishes the node's domain, characterized by two sets: λ_a, encompassing all packets permitted to traverse the node, and λ_d, representing those packets that are denied passage. In the absence of an ACL, the former set corresponds to the entirety of possible packets, while the latter remains empty.

- **Forwarding Table**: after a packet enters a node, it undergoes the switching operation and is forwarded to the appropriate output port based on the rules outlined in the forwarding table, another integral characteristic of each node.

- **Transformation Function** τ: as previously mentioned, within a network node, a packet may undergo various transformations. In addition to the forwarding table and the domains λ_a and λ_d, each network node incorporates another data structure responsible for determining whether a packet should undergo transformation and, if so, specifying the nature of the transformation. This functionality is captured by the transformation function τ, which is

designed with one or more input domains, each corresponding to one or more actions, and output domains.

Hence, it's imperative to represent the rules within the ACLs of nodes, the entries in the forwarding tables, and the domains of the transformation function using predicates, employing the same model utilized for network traffic. This approach enables comparison between the predicate describing incoming packets and the predicates characterizing each encountered node, facilitating decisions regarding forwarding and transformation behavior based on matches with the rules or domains of the node.

Different methods exist for modeling predicates, but for this thesis work, the chosen representation was introduced by [9] and involves implementing Binary Decision Diagrams (BDDs) in Java. BDDs are acyclic, directed, rooted graph structures utilized for representing Boolean functions. The approach for representing predicates introduced in [10] states that "a predicate is the conjunction of subpredicates, one for each packet field considered, and this conjunction is denoted by the tuple of its sub-predicates". This means that, when dealing with IP packets, they can be characterized as conditions defined over the IP quintuple – *IP Source*, *IP Destination*, *source port*, *destination port*, and *protocol type*. Each component in this set is further depicted by a sub-predicate describing the individual field. The overall condition representing an IP packet is formed by combining all these sub-conditions. It's worth noting that each sub-condition can represent a single value, a range of values, or even the entire range, denoted by the wildcard symbol "*".

3.6.2 Atomic Flows

The initial approach under consideration involves the usage of Atomic Predicates (APs), a concept introduced in 2015 by researchers to address the Network Reachability problem [11]. This concept has undergone modifications and adaptations to suit the challenges of verifying the satisfiability of NSRs and refining the problem within the VEREFOO framework, as proposed by [9].

In this solution, each intricate predicate is decomposed into a collection of simpler and minimal, unique, and fully representative APs. Subsequently, this set of is employed to generate a set of interesting flows within the network. These flows may exclusively consist of elements from the computed set of APs, representing the traffic between any two nodes.

Given a predicate P, the corresponding set of APs $A(\{P\})$ is computed through the following process:

$$A(\{P\}) = \begin{cases} \{true\}, if P = false or true \\ \{P, \neg P\}, otherwise \end{cases} \tag{3.1}$$

The *Atomic Flows* (AFs) approach leverages APs to delineate the characteristics of each traffic flow within the network. It involves configuring each firewall with rules expressed exclusively through APs. The process begins by identifying what

we term as "interesting" predicates, which are predicates linked to nodes associated with one of the given NSRs.

These "interesting" predicates encompass those representing source traffic and destination traffic for each requirement. Once the set of APs is computed, the next step is to generate all possible AFs for each user requirement. Then they are given in input to the MaxSMT solver or to the heuristic algorithm, depending on the version of VEREFOO we are considering, to allocate and configure the needed NSFs.

Using APs offers some advantages in network traffic modeling. Since predicates are unique, they can be associated with integer identifiers, simplifying solver's or heuristic's operations and improving performance. It is possible to work with integers as representations of traffic, leading to a more streamlined problem definition. Operations involving intersections and unions are less complex with integer sets compared to more intricate predicate representations.

Configuring NFs becomes easier with APs. Each configured rule is associated with a specific input traffic, and since APs are inherently disjointed, configured rules only affect specific portions of traffic without impacting others. However, working with integers may result in the inability to merge multiple rules, leading to a larger number of configured rules. Post-processing tasks can aggregate rules by converting integer identifiers back to IP-quintuples.

While the approach using AFs may not yield the absolute smallest number of configured rules, it achieves the smallest number of disjointed rules, representing a practical trade-off. A notable disadvantage is the computational intensity of the initial step, requiring the generation of APs from interesting predicates. This involves processing each interesting predicate, computing its intersection with all other APs, and adding it to the set, making the process time-consuming due to the need for disjointed traffics.

3.6.3 Maximal Flows

The second model, known as *Maximal Flows* (MFs), is discussed in more details in [12]. In contrast to the preceding solution, where the aim was to break down traffic flows into smaller components, achieving the level of granularity but also a higher count of flows, the MFs approach takes a different direction. Here, the objective is to reduce the overall number of generated flows by consolidating various sub-flows into a single MF. This aggregated flow remains representative of all the individual sub-flows it encompasses.

In this method, multiple traffic flows that are merged into the same MF must have an identical behavior as they traverse through different network nodes. This consolidation results in a larger granularity and a reduced number of flows that effectively represent the entire network. Similar to the previous case, traffic flows are represented as a series of alternating nodes and predicates. However, unlike before, the predicates used here are not atomic; rather, they articulate the combination of multiple IP quintuples through disjunction. The set of MFs F_r^M is defined as a subset of F_r, containing only those flows which are not subflows of other flows in F_r.

All flows exhibiting similar behavior are grouped together into a single MF. Following this, either the MaxSMT problem or a heuristic algorithm is formulated using only the set F_r^M. Despite its reduced size compared to F_r, F_r^M retains the same level of expressiveness. The key benefit of this method lies in the considerably faster computation of F_r^M compared to the set of AFs. This efficiency is attributed to the fact that, in contrast to the previous method, the algorithm for computing MF does not require any initial computation time for traffic flow computation.

However, a notable disadvantage emerges in that the traffic exchanged between nodes for each MF is neither disjointed nor unique. Consequently, it cannot be associated with a simple integer identifier, as could be done with APs. The solver is forced to operate with a representation of the predicate. This solution necessitates a total of 13 fields: 4 integers for the source IP address, 4 integers for the destination IP address, 2 integers for the range of source ports, 2 integers for the range of destination ports, and finally, a string for the protocol type. The increased number of variables provided as input to the solver significantly impacts the final resolution performance.

Chapter 4

The MaxSMT Problem

This chapter explains the initial approach taken in shaping the VEREFOO framework. It relies on a theorem prover, addressing the MaxSMT problem for an optimal solution.

4.1 Maximum Satisfiability Modulo Theories

4.1.1 Boolean Satisfiability Problem (SAT)

In the realms of logic and computer science, the Boolean Satisfiability problem, commonly known as SAT, poses a fundamental question: is there a valid interpretation for a given Boolean formula? This intricate problem revolves around determining whether it's possible to substitute the variables in a given Boolean formula with TRUE or FALSE values in such a way that the formula evaluates to TRUE. When such a consistent assignment exists, the formula is said to be satisfiable; otherwise, if no such assignment is possible, the formula is considered unsatisfiable. It's crucial to note that, in the context of SAT, the focus is not on looking for the best solution, but it is sufficient to find a combination of variables such that the formula is TRUE. The goal here is to detect if the given formula is satisfiable or not.

For instance, consider the formula "a AND NOT b." This formula is satisfiable because assigning the values a = TRUE and b = FALSE results in the formula evaluating to TRUE. Conversely, the formula "a AND NOT a" is unsatisfiable, as no assignment of values exists that would make the formula TRUE for all possible variable combinations.

The significance of SAT goes beyond its inherent complexity; it was the first problem proven to be NP-complete, as established by the Cook–Levin theorem. This classification implies that solving any problem within the complexity class NP (nondeterministic polynomial time), which includes a wide range of natural decision and optimization problems, is no more difficult than solving SAT. Despite its importance, there is currently no known algorithm that efficiently solves all SAT problems. The quest for such an algorithm remains a complex challenge, closely

tied to the unresolved P versus NP problem, a prominent and open question in the realm of computing theory.

4.1.2 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories, also known as SMT, is a generalization of SAT. While the SAT problem's formulas are expressed by means of the classic simple boolean logic, the SMT problem uses more complex formulas involving real numbers, integers, and various data structures like lists, arrays, bit vectors, and strings. The term "modulo" signifies that these expressions are interpreted within a specific formal theory in first-order logic, often without incorporating quantifiers.

SMT solvers, such as Z3 and cvc5, serve as instrumental tools designed to solve the SMT problem for a practical subset of inputs. These solvers are used as a building block for applications across diverse domains in computer science, including automated theorem proving, program analysis, program verification, and software testing.

Given that Boolean Satisfiability is already known to be NP-complete, the SMT problem, thanks to its more complex language, leads to the creation of much more intricate problems. In fact, a SMT problem is typically NP-hard, and in numerous theories, it is undecidable. An undecidable problem is a decision problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes-or-no answer. Researchers delve into identifying theories or subsets of theories leading to a decidable SMT problem and study the computational complexity of such decidable cases. The resulting decision procedures are frequently implemented directly in SMT solvers.

4.1.3 Maximum Satisfiability Modulo Theories (MaxSMT)

Maximum Satisfiability Modulo Theories, or MaxSMT, is itself a generalization of SMT. It is essential in scenarios where finding the optimal assignment of truth values is critical for decision-making processes. It is widely used in applications such as formal verification of hardware and software systems, automated theorem proving, and constraint optimization problems.

While SAT and SMT focus on determining the satisfiability of logical formulas, MaxSMT goes a step further by aiming to find an assignment of truth values to variables that maximize the number of satisfied clauses or constraints of which the problem is composed. The fundamental components of the MaxSMT problem include a set of logical constraints and an objective function to be maximized. These constraints typically represent real-world conditions, and the objective function encapsulates the optimization goal. In essence, MaxSMT navigates the intricate landscape of logical possibilities to identify the most favorable configuration that simultaneously satisfies the specified constraints and maximizes the defined objective.

For example, the following conjunctive formula which is made by 2 variables and the simple boolean operators AND, OR and NOT:

$$(x_0 \lor x_1) \land (x_0 \lor \neg x_1) \land (\neg x_0 \lor x_1) \land (\neg x_0 \lor \neg x_1)$$

is inherently unsatisfiable. Regardless of the assigned truth values to its two variables, at least one of its four clauses will inevitably evaluate to false. However, when considering it as an instance of the MaxSMT problem, the goal is to maximize the number of satisfied clauses. In this specific case, every truth assignment results in three out of four clauses being true. Hence, the optimal solution to the MaxSMT problem for this formula is three.

In order to make the formulation of a MaxMST problem more flexible, there are some variants of it. The idea is that sometimes not all constraints of a problem can be satisfied. To address this, the problem is categorized into two groups: *hard constraints* (or clauses) that must be satisfied and *soft constraints* that are optional. The input NSRs and additional constraints set by the service designer, such as firewall placement or restrictions on service function allocation, are represented using hard constraints. Conversely, soft constraints are employed for optimization objectives. In this framework, the solver aims to select the solution that satisfies all hard constraints while maximizing the total value of satisfied soft constraints.

When considering soft constraints, varying weights can be assigned to clauses to represent the penalty incurred if a clause is falsified, reflecting the relative importance of different constraints. By introducing weights to clauses, the instance becomes **weighted**, and the classification into hard and soft clauses makes the instance **partial**. For a given **weighted partial MaxSMT** instance the objective is to identify an assignment that satisfies the hard clauses while minimizing the cumulative weight of falsified clauses. This approach is used in VEREFOO to optimize resource consumption, particularly in terms of the total number of allocated firewalls and configured rules, as shown in [13]. This weighted partial MaxSMT variant enables automation, optimization, and formal correctness within VEREFOO, requiring minimal human intervention beyond providing security policies. Moreover, the approach is optimized, given that the soft constraints articulate the optimization objectives, and formally correct, since the requirements are represented with the hard constraints.

4.2 Z3 Theorem Prover

Researchers and practitioners often leverage advanced tools called solvers to address MaxSMT problems efficiently. These solvers use sophisticated algorithms and techniques to explore the solution space and their primary goal is to identify the assignment that achieves the maximum satisfaction under the given constraints. One notable example of such a solver is Z3, developed by Microsoft Research [14]. It is a software that has the ability to automatically solve satisfiability problems and prove theorems in first-order logic.

One of the main features of Z3 is its ability to support various logics, making it flexible and suitable for a wide range of applications. This includes logics such as first-order logic, differential logic, array logic, and many others.

4.2.1 Z3 Architecture

Figure 4.1 illustrates the main components of Z3.

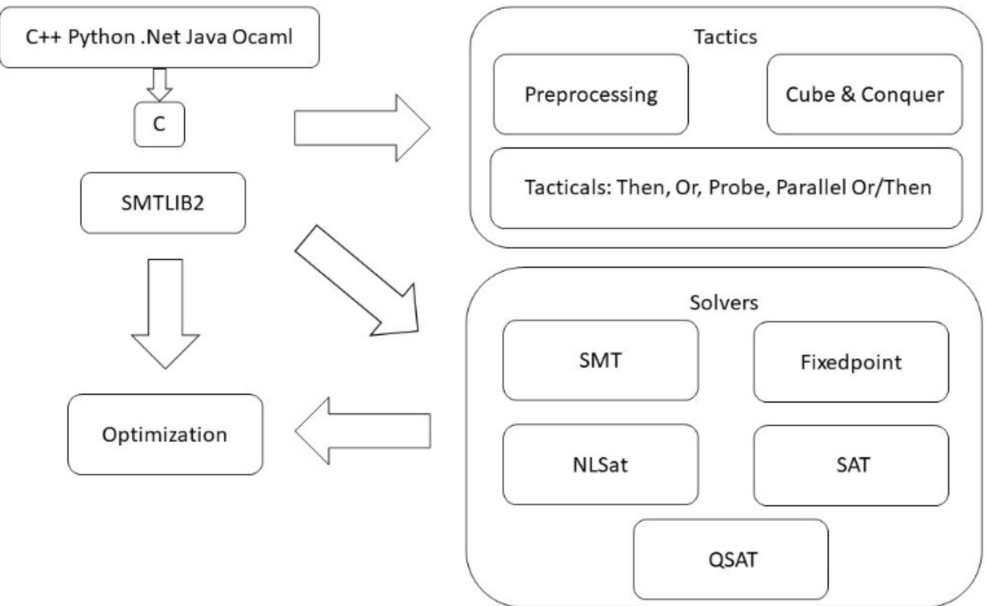


Figure 4.1: Overall system architecture of Z3

- The top left summarizes the interfaces to Z3. One can interact with Z3 over SMT-LIB2 scripts, which is the default input format for Z3. They can be supplied as a text file or pipe to Z3, or using API calls from a high-level programming language (in the case of VEREFOO, Java APIs are used) that are proxies for calls over a C-based API.
- In contrast to solvers that ultimately check the satisfiability of a set of assertions, tactics transform assertions to sets of assertions, in a way that a proof-tree is comprised of nodes representing goals, and children representing subgoals. Many useful pre-processing steps can be formulated as tactics. They take one goal and create a subgoal.
- Solvers maintain a set of formulas and supports satisfiability checking, and scope management: Formulas that are added under one scope can be retracted when the scope is popped.
- Z3 provides support for optimizing objective functions in addition to determining satisfiability. This feature is valuable in scenarios where retrieving optimal models based on certain criteria is necessary. The optimization module in Z3 can handle various objective functions by specifying whether to maximize or minimize the values of a particular arithmetical term t. For instance, using the objective *maximize(t)* instructs the solver to find solutions that maximize the value of t. Alternatively, the weighted partial MaxSMT approach, as explained in Section 4.1, offers another method to specify optimization objectives.

4.2.2 Z3 Example

The following is an example of how Z3 works. In particular, Listing 4.1 is the description of the problem to solve, while Listing 4.2 illustrates the solution.

```
(declare-const a Int)
(declare-const b Int)
(assert (= (+ a b) 20))
(assert (= (+ a (* 2 b)) 10))
(check-sat)
(get-model)
```

Listing 4.1: Example of a MaxSMT problem expressed in Z3 language

```
sat
(model
  (define-fun b () Int
    -10)
  (define-fun a () Int
    30)
)
```

Listing 4.2: Example of a SAT solution expressed in Z3 language

Chapter 5

Heuristics in VEREFOO

In its initial version, VEREFOO was developed using the MaxSMT problem approach, ensuring an optimal solution to the problem. This methodology aims to configure the minimum number of firewalls in the input topology and implement the minimum number of rules on them to meet the user-specified security requirements. However, since the MaxSMT problem is NP-complete, meaning there is no efficient (polynomial) algorithm to solve the problem in general, the resolution time increases non-linearly with the complexity of the input topology and the set of requirements to be satisfied. From subsequent tests, which will be presented later, it was discovered that the increase in time is not linear but rather exponential. For this reason, a decision was made to develop a second version of the framework based on a heuristic algorithm, which will be described in this chapter.

In VEREFOO, two different types of heuristic algorithms can be employed:

1. **complete heuristic** is a type of heuristic algorithm that terminates only when all isolation requirements have been satisfied. It is described in Section 5.2;

2. **partial heuristic** instead is an algorithm that, rather than terminating the heuristics when all the isolation requirements are satisfied, alternative termination conditions are introduced. In this case, only a part of them is satisfied. It is described in Section 5.3.

However, before delving into these methods, Section 5.1 provides a description of what heuristics entails.

5.1 The Heuristic Approach

The word "heuristic" derives from the ancient Greek *heurískō*, meaning "to discover" or "to find." In a broader and more general sense, the term refers to "that part of epistemology and scientific method that aims to foster the search for new theoretical developments, empirical discoveries, and technologies. It involves an

approach to problem-solving that does not follow a clear path but relies on intuition and the temporary state of the surroundings to generate new knowledge" [15]. Heuristics can be seen as a way of proceeding "outside the box." Where rigid and well-defined methodologies fail to lead to problem resolution or only solve it in specific scenarios, heuristics suggest an alternative approach to the classical view. However, this does not imply completely abandoning Galilean methodological rigor or entirely discarding the widely employed scientific method up to now. Instead, it adopts flexibility to seek alternative solutions to those previously employed.

The key to understand heuristic thinking lies in a specific point of the above definition: "that does not follow a clear path but relies on intuition and the temporary state." This phrase should be understood as follows: when classic techniques and methodologies do not yield useful results, the analysis and observation of the studied phenomenon should serve as input for the intuition of alternative paths. This provides a different perspective from the one adopted so far, aiming to quickly offer a solution to the problem, given the temporary state on which heuristics is based.

5.2 Complete Heuristics

The heuristic algorithm must balance various and often conflicting objectives.

- Our primary focus lies in achieving a delicate balance between **minimizing allocation costs and maximizing filtering performance**. This entails the objective of deploying the fewest number of firewalls with the least number of rules, mirroring the principles underlying the MaxSMT problem. It's important to note that, given the heuristic nature of the algorithm, we aim for a high level of efficiency without necessarily achieving the same level of optimality.
- Simultaneously, we want the **maximization of network security and throughput**. To enhance security, our strategy involves blocking traffic as close to the source as possible. However, this approach may potentially impact allocation costs, as it might necessitate the deployment of multiple firewalls.

This challenge requires the implementation of an intelligent heuristic algorithm that optimizes resource allocation, minimizing the number of firewalls and rules to ensure robust defense and high operational efficiency.

Similar to the MaxSMT-based version, the heuristic algorithm-based approach also requires two inputs: the *Service Graph* and a set of *Network Security Requirements* that need to be satisfied. However, in this case, a NSR r can be of 3 different types:

1. **isolation**, if all the traffic flows satisfying r must be blocked;
2. **complete reachability**, if all the traffic flows satisfying r must not be blocked;

3. **partial reachability**, if there must exist at least a network path where all the traffic flows satisfying r are not blocked.

The set R of all the NSRs is defined as:

$$R = R^i \cup R^{cr} \cup R^{pr}$$

where

- R^i is the set of all the *isolation* requirements;
- R^{cr} is the set of all the *complete reachability* requirements;
- R^{pr} is the set of all the *partial reachability* requirements.

The heuristic algorithm can be delineated into three primary steps.

1. **Initialization**: at this stage, which is better described in Subsection 5.2.1, the algorithm goes through the initial setup and prepares for subsequent phases. The fundamental steps of this phase include the **computation of weights and predicates**. Weight calculation means assigning a *weight* to each Allocation Place after transforming the Service Graph into the Allocation Graph. This weight is based on some criteria, like the number of flows passing through it and the type of requirements associated with those flows. The weights will be utilized in the subsequent phase. Additionally, during this stage, *predicate velues* are computed using traffic flows models such as Maximal Flows or Atomic Predicates, as described in Chapter 3.
2. **Allocation**: in this step, described in Subsection 5.2.2, leveraging the information generated during the initialization phase, the algorithm determines the appropriate allocation places to minimize the number of firewalls needed to meet the requirements.
3. **Configuration**: after all the firewalls have been allocated in the allocation stage, in this phase, described in Subsection 5.2.3 it is decided which rules must be configured on each firewall.

5.2.1 Initialization

This phase begins after the input SG has been transformed into an AG. The purpose of this stage is to assign a weight to each generated Allocation Place and then create an ordered list where the Allocation Places are arranged in descending order of weight. However, to assign the weights, given a set of NSRs R, it is necessary to first calculate all the flows F_R that satisfy those requirements. The algorithm for *Maximal Flows* computation or the algorithm for the *Atomic Predicates* computation can be used to populate the F_R set. This algorithm allows to compute several pieces of information (e.g., relationships between requirements and flows, paths crossed by the flows), avoiding their computation in later stages.

In particular, some useful informations that are exploited in subsequent phases can be retrieved after computing the traffic flows:

1. $\pi : F_R \rightarrow (N_A)^*$ maps a flow $f \in F_R$ to the ordered list of nodes crossed by f;
2. $\rho : F_R \rightarrow R$ maps a flow $f \in F_R$ to the requirement $r \in R$ whose conditions are satisfied by f;
3. $\phi : R \rightarrow \mathcal{P}(F_R)$ maps a requirement $r \in R$ to the set of flows that satisfy its conditions;
4. $\Pi : R \rightarrow \mathcal{P}((N_A)^*)$ maps a requirement $r \in R$ to the set of paths crossed by at least a flow $f \in F_R$ satisfying the conditions of r.

The heuristics is based on three main predicates:

1. **blocked**: $F_R \rightarrow \mathbb{B}$ maps a flow $f \in F_R$ to true if the flow is blocked before reaching the destination, to false otherwise;
2. **satisfied**: $R \rightarrow \mathbb{B}$ maps a requirement $r \in R$ to true if its conditions are satisfied, to false otherwise;
3. **allocated**: $A_A \rightarrow \mathbb{B}$ maps an AP $a \in A_A$ to true if a firewall is allocated in a, to false otherwise.

Once they have been generated, they must be initialized, so we must provide a value for each flow $f \in F_R$, for each requirement $r \in R$ and for each Allocation Place $a \in A_A$. In particular, in the initial situation we have that:

- no traffic flow is **blocked**, because no firewall is allocated in the network and any packet can reach any destination starting from any source:

$$\forall f \in F_R.blocked(f) = false \tag{5.1}$$

- all the *isolation requirements* are **not satisfied** because the related flows can reach the destinations, while all the *reachability requirements* are **satisfied** because their related flows are not blocked:

$$\forall r \in R.satisfied(r) = \begin{cases} false \text{ if } r \in R^i \\ true \text{ if } r \in R^{cr} \vee r \in R^{pr} \end{cases} \tag{5.2}$$

- no firewall is **allocated** in any Allocation Place:

$$\forall a \in A_A.allocated(a) = false \tag{5.3}$$

Subsequently, the following step involves computing and assigning a **weight** w_a to each Allocation Place $a \in A_A$. But, before doing this operation, it is needed to perform the same operation for each flow. This is the step that encapsulates the optimization criteria of the methodology. In the MaxSMT-based version of the framework, this task was handled by incorporating soft clauses into the formulation of the MaxSMT problem. Conversely, the heuristic algorithm employs an ordered list where all Allocation Places are arranged in descending order based on their weights.

The computation of w_a must consider:

1. the number of *flows* crossing a, and the **type** of requirements associated to them;

2. the **position** of a in each path $\pi(f)$ for each flow f crossing a.

Let $\pi(f)$ represent the ordered list of nodes traversed by flow f, destination included but not the source.

These factors should be possibly mediated, because not all requirements have the same level of importance.

However, to compute the weights of the Allocation Places, it is first necessary to calculate and assign a weight to each flow $f \in F_R$. This is because there might be flows in the network that are not relevant for meeting the requirements, thus influencing the allocation places associated with them.

$$\forall f \in F_R.w_f = \begin{cases} +1 & \text{if } \rho(f) \in R^i \\ -1 & \text{if } \rho(f) \in R^{cr} \\ -\frac{1}{|\Pi(\rho(f))|} & \text{if } \rho(f) \in R^{pr} \end{cases} \tag{5.4}$$

There are three cases, one for each type of requirement:

- Case 1: $\rho(f) \in R^i$

  Each flow f such that $\rho(f) \in R^i$ must be blocked to have isolation. Therefore, the weight w_f is positive and unitary because the level of "importance" of the flow is high since it is needed to allocate a firewall to satisfy the requirement.

- Case 2: $\rho(f) \in R^{cr}$

  Each flow f such that $\rho(f) \in R^{cr}$ must not be blocked to have complete reachability. Therefore, the weight w_f is negative and unitary because the level of "importance" of the flow is lower since the requirement is satisfied yet and it is not needed to allocate a firewall for it.

- Case 3: $\rho(f) \in R^{pr}$

  It is enough that all the flows crossing a path among the $|\Pi(\rho(f))|$ paths are not blocked to have partial reachability. Therefore, the weight w_f is negative and between 0 and 1. Its value is equal to $-\frac{1}{|\Pi(\rho(f))|}$.

Now, each $f \in F_R$ has an associated weight w_f. This parameter is used in the subsequent step, to compute Allocation Places weights through the following formula:

$$\forall a \in A_A.w_a = \Big[\sum_{f \in F_R | c_1(f)} m_f \alpha w_f\Big] + \Big[\sum_{f \in F_R | c_2(f)} m_f (1-\alpha)(1 - \frac{i_{af}}{|\pi(f)|})\Big] \tag{5.5}$$

where:

- i_{af} is an **index weight** that is computed for a pair $a \in A_A$ and $f \in F_R$ such that $a \in \pi(f)$. It identifies the position of a in $\pi(f)$, with index starting from 0. For example, let's consider the AP:

$$a_8$$

and a path:

$$\pi(f) = [e_2, a_1, s_5, a_8, s_1 1, a_1 5, e_6]$$

where e_x is an end point, s_x is a switch and a_x is an AP. The index weight for a_8 is:

$$i_{a_8 f} = 3$$

- α is a parameter that ranges from 0 to 1 included. If it is equal to 0, only "max-security objective is considered". If instead it is equal to 1, only "min-allocation" objective is considered. By modifying this parameter, we can customize the behavior of the framework according to our needs.

- m_f is a parameter that for now is set to 1, but it may be set to 0 later, when a flow must **not be considered** anymore in the sum.

In the summations there are also two conditions:

- $c_1(f) = a \in \pi(f)$: the AP a is part of the list of nodes $\pi(f)$;

- $c_2(f) = a \in \pi(f) \wedge \rho(f) \in R^i$: as condition $c_1(f)$, but f must be involved by an isolation requirement. It means that the second summation is considered only for isolation requirements.

At this point, all the APs of the A_A set have a weight associated and are introduced in the L_A list. An example is showed in Figure 5.1. Initially, there is no ordering in L_A, but this list is useful for the heuristic decision on which AP a firewall should be allocated. When an AP will be removed from L_A, it means that a firewall is allocated in it.

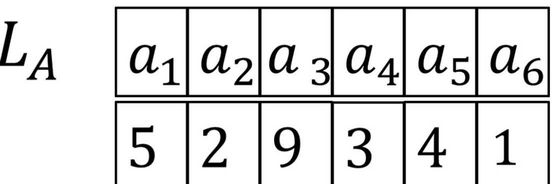


Figure 5.1: Example of list L_A. The first row is the list of APs, with each of them being associated with a corresponding weight listed in the second row

5.2.2 Allocation

At this point, the initialization step has produced an unordered list of APs. The first step of the second phase, which is the Allocation step, involves sorting this list in descending order of w_a weight: the top positions will include all APs with higher weights, while the lower positions will contain those with lower weights. Figure 5.2 depicts the same list shown in Figure 5.1 after the sorting process.

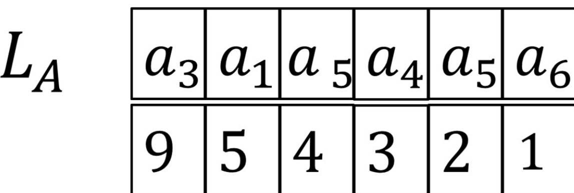


Figure 5.2: Example of ordered list L_A

Then, the element a^* having highest weight w_a in the L_A list is extracted. This AP is decided to be a position where a firewall must be allocated.

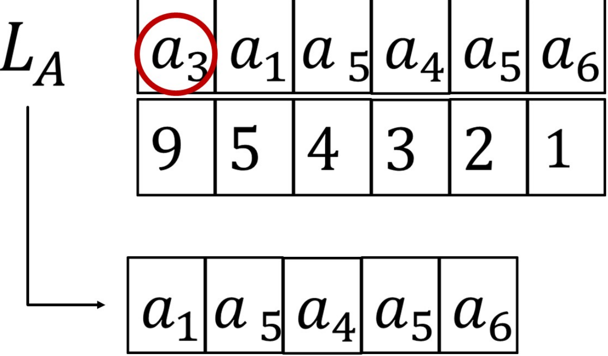


Figure 5.3: Allocation Place a_3 is extracted from list L_A

In this stage of the algorithm, the firewall that is allocated is characterized only by a *DENY* action. In fact, the primary objective is to have all the firewalls that are really needed to enforce isolation requirements, which are the most important. The impact of the reachability requirements is evaluated in a subsequent stage. It's important to note that this is only a temporary configuration: later in the process, each allocated firewall may have a different default action and will be configured so as to enforce all the requirements.

After the decision of allocating a firewall in a^*, the values that have been described in Subsection 5.2.1 must be updated:

- the allocated predicate returns *true* when applied to a:

$$allocated(a^*) = true$$

- all the traffic flows crossing a are blocked:

$$\forall f \in F_R.(a^* \in \pi(f) \rightarrow blocked(f) = true) \tag{5.6}$$

- a flow associated to an isolation requirement that is blocked must not be considered anymore in the w_a computation (because there is already a firewall responsible for blocking it):

$$\forall f \in F_R.(a^* \in \pi(f) \land \rho(f) \in R^i \rightarrow m_f = 0) \tag{5.7}$$

Additionally, all the flows blocked by a firewall allocated in a are stored in a set identified by the B_{a^*} symbol.

There are then some updates which are related to requirement satisfaction.

- An isolation requirement $r \in R^i$ is satisfied if all the traffic flows satisfying r are blocked:

$$\forall r \in R^i.((\forall f \in \phi(r).blocked(f) = true) \rightarrow satisfied(r) = true) \tag{5.8}$$

- A complete reachability requirement $r \in R^{cr}$ is not satisfied if there exists a traffic flow satisfying r that is blocked:

$$\forall r \in R^{cr}.((\exists f \in \phi(r).blocked(f) = true) \rightarrow satisfied(r) = false) \tag{5.9}$$

- A partial reachability requirement $r \in R^{pr}$ is not satisfied if there exists not a path where all the traffic flows satisfying r are allowed:

$$\begin{aligned} \forall r \in R^{pr}.(.(\nexists l \in \Pi(r)(\forall f \in \phi(r).l = \pi(f).blocked(f) = false)) \\ \rightarrow satisfied(r) = false) \end{aligned} \tag{5.10}$$

The allocation algorithm concludes if all the isolation requirements are satisfied:

$$\forall r \in R^i.(satisfied(r) = true) \tag{5.11}$$

If this condition is false, the following steps are performed:

1. for each $a \in A_A$ that is still present in the L_A list, the w_a weight is recomputed, in light of the updates related to the m_f coefficients in Formula 5.5;
2. the procedure is repeated from the *allocation* step, when the list is re-ordered.

5.2.3 Configuration

Once condition 5.11 is satisfied, the algorithm enters its final phase, known as the *configuration* phase. In the *Allocation* phase, firewalls are allocated to the appropriate APs. However, all these firewalls are configured with a temporary default *deny* rule, ensuring compliance with isolation requirements, but not with the reachability ones.

Now, an analysis of the rules which should be defined in a firewall allocated in the AP $a \in A_A$ is performed, as if it did not have a default action. This leads to create two sets:

1. $\mathbb{F}_a^D$ is the set of the denying rules deriving from isolation requirements;
2. $\mathbb{F}_a^A$ is the set of the allowing rules deriving from complete and partial reachability requirements.

As mentioned earlier, the set B_a encompasses all the flows blocked by a firewall assigned to the AP a. Given that, as of this stage, all the firewalls are configured with a temporary default *DENY* rule. It means that the B_a set for each firewall includes all flows traversing it. Subsequently, for each flow within the B_a set, a rule is formulated. The determination of whether the rule is of type *ALLOW* or *DENY* depends on the source of the flow. The preocess is explained blow.

- Considering each $a \in A_A$ such that $allocated(a) = true$, a denying rule is computed for a firewall in a for each flow appearing in the corresponding B_a set and deriving from an isolation requirement:

$$\begin{aligned} \forall a \in A_A | allocated(a) = true. \forall f \in B_a | \\ \rho(f) \in R^i.((\tau(f, a), deny) \in \mathbb{F}_a^D) \end{aligned} \tag{5.12}$$

- Considering each $a \in A_A$ such that $allocated(a) = true$, an allowing rule is computed for a firewall in a for each flow appearing in the corresponding B_a set and deriving from an complete reachability requirement:

$$\begin{aligned} \forall a \in A_A | allocated(a) = true. \forall f \in B_a | \\ \rho(f) \in R^{cr}.((\tau(f, a), allow) \in \mathbb{F}_a^D) \end{aligned} \tag{5.13}$$

- Considering each $r \in R^{pa}$ such that $satisfied(r) = false$, among all the node lists appearing in the $\Pi(r)$ set, a list $l*$ is selected as the list having the lowest number of APs where firewalls are allocated and, with parity of value, having the lowest overall length. For each $a \in l^*$ such that $allocated(a) = true$, an allowing rule is computed for each flow deriving from r:

$$\begin{aligned} \forall r \in R^{pa}. \forall f \in \phi(r) | \pi(f) = l^*. \forall a \in A_A | \\ allocated(a) = true.((\tau(f, a), allow) \in \mathbb{F}_a^D) \end{aligned} \tag{5.14}$$

5.3 Partial Heuristics

There are some cases in which, rather than having a complete heuristics, we would like to have a partial heuristic.

The idea is that, especially when dealing with a network with limited resources and not overly strict security constraints, instead of concluding the heuristics when all isolation requirements are met, alternative termination conditions could be considered. For instance:

- only a certain percentage of requirements needs to be satisfied;

- only a specific number of APs must be allocated.

Finally, VEREFOO is executed so that it can compute the allocation scheme and configuration for the remaining APs. In case the execution fails, the last firewall allocated is removed, and VEREFOO is re-executed with a new free AP.

Other possible optimizations conditions may be:

- remove all the APs that are not crossed by any flow related to unsatisfied isolation requirements. This means to reduce the number of APs that are considered during the weight computation, making the operation more efficient. This is possible because in the *Allocation* stage, the values of the *satisfied* predicates were always updated;
- remove a certain number of APs having the lowest weights. The idea is similar to fixing firewalls in APs with the highest weights. However, this could lead to problem unfeasibility, since some requirements would remain unsatisfied. In that case, VEREFOO may require to be re-executed after reintroducing an AP at a time.

However, this thesis focuses on the complete heuristics, while the partial heuristics might be an idea for a future work.

5.4 Alternative idea for Heuristics

The *Configuration* stage may be solved with a Branch-and-Bound method. This method allows to reach the optimal solution by exploring the full solution space, and this means there is not anymore a heuristic algorithm, but a solution more similar to the resolution of the MaxSMT problem.

This solution may become heuristic by introducing some **bounds** in the exploration. For example, a certain bound of the ratio

$$\frac{\#allocatedfirewalls}{sumoftheirweights}$$

is estabilished and some branches are cut earlier than their end.

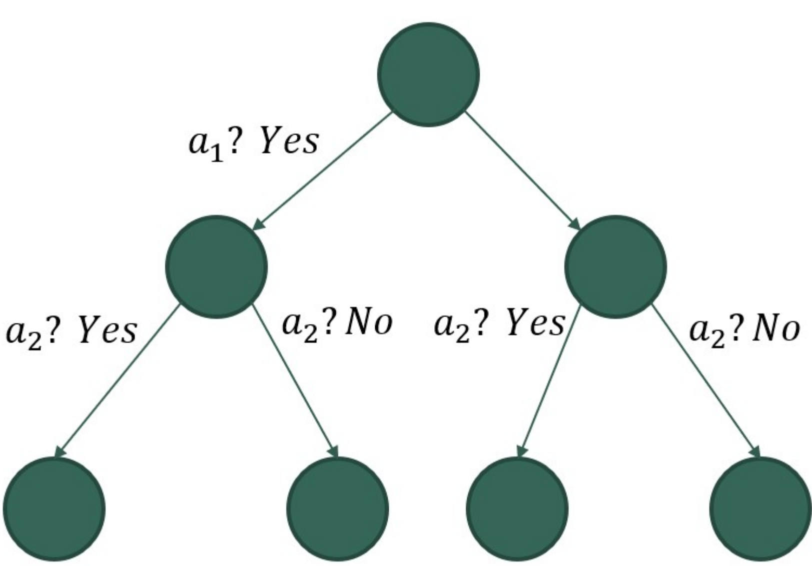


Figure 5.4: Visual example of the Branch-and-Bound method

Chapter 6

Test Campaign Preparation: Code Interventions and New Topology Design

6.1 VEREFOO Implementations

The VEREFOO framework, whose code is publicly available on GitHub [16], has evolved over time through various versions. This chapter focuses on four of these versions, with two based on the MaxSMT problem discussed in Chapter 4 and the other two on the heuristic algorithm described in Chapter 5. In fact, one of the objectives of this thesis is to examine the performance and optimization differences between the MaxSMT-based versions and those based on the heuristic algorithm, as well as to evaluate the implementations for modeling traffic flows: Atomic Predicates and Maximal Flows described in Chapter 3.

The following versions are considered, using the names of their corresponding branches on the GitHub repository:

- **BudapestWithAP**: This version addresses the MaxSMT problem, employing the Atomic Predicates algorithm for traffic flows modeling.
- **MaximalFlowsZ3**: similar to BudapestWithAP, this version tackles the MaxSMT problem but uses the Maximal Flows algorithm for traffic flow modeling.
- **HelsinkiWithAP**: this version adopts the heuristic algorithm, using Atomic Predicates for traffic flow computation.
- **HelsinkiWithMF**: Also based on the heuristic algorithm, this version employs the Maximal Flows approach for traffic flow computation.

Before effectively starting the testing campaign aimed at highlighting the differences between these versions, some modifications were made to the VEREFOO code regarding the heuristics-based ones. These modifications have been detailed in the following section.

6.2 Code Interventions

6.2.1 Allocation Places List Sorting

The first correction made to the code pertains to the section within the heuristic algorithm where the list of APs is sorted based on their assigned weight. Specifically, the code related to this operation was as follows:

```
Collections.sort(allocationPlaces, new Comparator<AllocationNode>() {
    @Override
    public int compare(AllocationNode ap1, AllocationNode ap2) {
        float ca1 = ap1.getAllocationWeight();
        float ca2 = ap2.getAllocationWeight();

        if (ca1 == ca2)
            return 0;
        else if (ca1 > ca2)
            return 1;
        else
            return -1;
    }
});
```

Listing 6.1: Sorting algorithm for Allocation Places list.

The code performs a classic sorting operation on a list of *AllocationNode* objects (which are the APs) named *allocationPlaces*. It uses the *Collections.sort()* method to sort the objects based on their allocation weight (*allocationWeight*).

The crucial part of the code is the definition of a new *Comparator<AllocationNode>* object inside the *sort()* method. This object is used to compare two *AllocationNode* objects and determine their relative order in the sorting.

The comparison logic is implemented in the *compare()* method. In this method, the allocation weights (*AllocationWeight*) of the two objects *ap1* and *ap2* are compared. If the weight of *ap1* is less than that of *ap2*, then *ap1* is considered "lesser" and will be positioned before *ap2* in the sorting. If the allocation weights are equal, the method returns 0, indicating that the objects are considered equivalent for sorting purposes. If the weight of *ap1* is greater than that of *ap2*, *ap1* is considered "greater" and will be positioned after *ap2* in the sorting.

The bug consisted in the fact that in this way, the list of APs was sorted in reverse order, i.e., in ascending order: APs with lower weights were placed at the beginning of the list, while those with higher weights were placed at the end. This meant that the first APs used to allocate the firewalls were the ones with the lowest weights, and therefore the contribution given by these firewalls was marginal. Despite this error, the framework still worked, but inevitably, a greater number of firewalls was allocated, and a greater number of rules was configured.

To fix the bug, it was sufficient to simply reverse the list's sorting order, as shown in Listing

```
        if (ca1 == ca2)
            return 0;
        else if (ca1 > ca2)
            return -1;
        else
            return 1;
        }
    });
```

Listing 6.2: Fixed sorting order of Allocation Places' list.

Below is an example demonstrating the effects of the previous sorting method and the current one. Let's consider a very simple Service Graph, like the one shown in Figure 6.1.

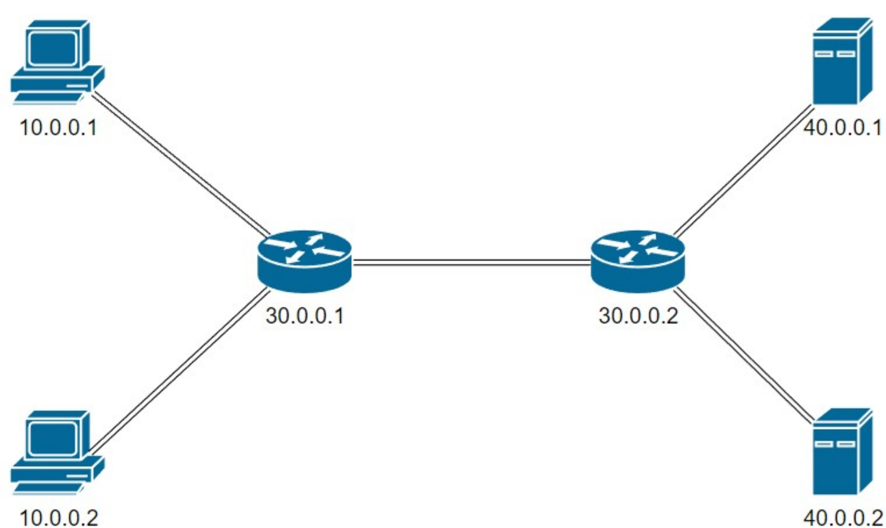


Figure 6.1: Example of a simple Service Graph

Then, Table 6.1 represents a set of NSRs that must be enforced.

Type	IPSrc	IPDst	portSRC	portDST
Isolation	10.0.0.2	40.0.0.1	*	*
Isolation	40.0.0.1	10.0.0.1	*	*
Reachability	10.0.0.1	10.0.0.2	*	*
Reachability	40.0.0.1	10.0.0.2	*	*

Table 6.1: Network Security Requirements of the example

Using the incorrect sorting method for the list of APs yields the result shown in Figure 6.2.

It can be noted that 2 firewalls have been allocated by the framework.

Table 6.2 shows the configuration of the firewall whose IP address is 20.0.0.1.

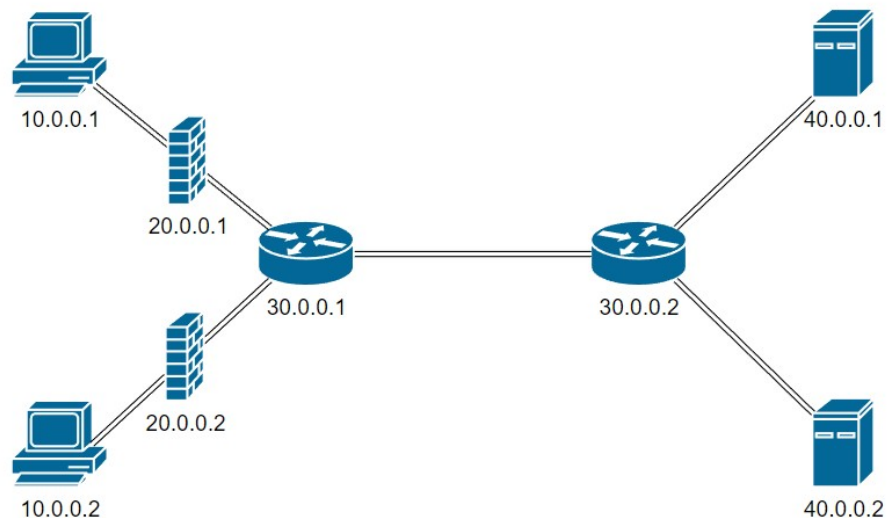


Figure 6.2: Outcome of the framework using the wrong sorting method

Number	actionType	IPSrc	IPDst	portSrc	portDst
1	DENY	40.0.0.1	10.0.0.1	*	*
2	ALLOW	10.0.0.1	10.0.0.1	*	*

Table 6.2: Configuration of firewall 20.0.0.1

Instead, Table 6.3 shows the configuration of the firewall whose IP address is 20.0.0.2.

Number	actionType	IPSrc	IPDst	portSrc	portDst
1	DENY	10.0.0.2	40.0.0.1	*	*
2	ALLOW	10.0.0.1	10.0.0.2	*	*
3	ALLOW	40.0.0.1	10.0.0.2	*	*

Table 6.3: Configuration of firewall 20.0.0.2

In the end, a total of 2 firewalls have been allocated and a total of 5 rules have been configured to enforce the NSRs showed in Table 6.1.

Now, the same SG showed in Figure 6.1 is considered, but it is used the right version of the sorting method for the list of APs.

Figure 6.3 shows the outcome.

Finally, Table 6.4 shows the configuration of the single Firewall which has been allocated.

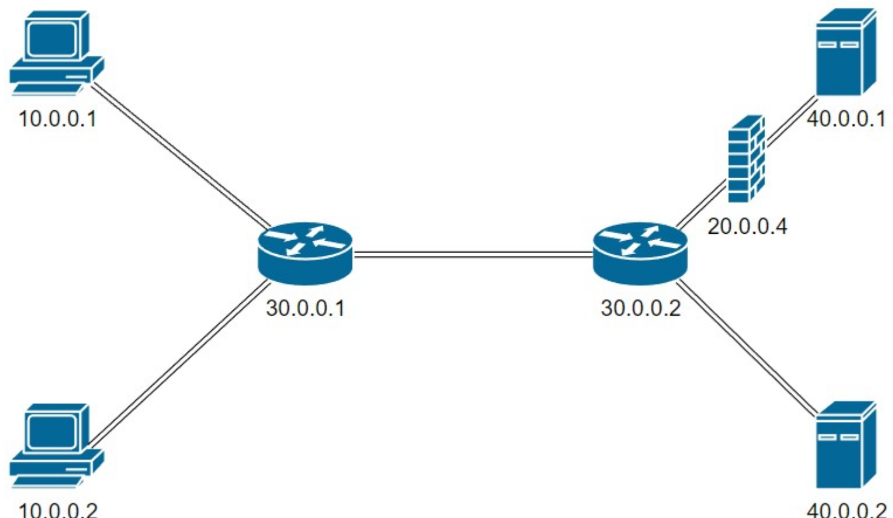


Figure 6.3: Outcome of the framework using the right sorting method

Number	actionType	IPSrc	IPDst	portSrc	portDst
1	DENY	10.0.0.2	40.0.0.1	*	*
2	DENY	40.0.0.1	10.0.0.1	*	*
3	ALLOW	40.0.0.1	10.0.0.2	*	*

Table 6.4: Configuration of Firewall 20.0.0.4

Considering the right sorting method, only 1 Firewall has been allocated and a total of 3 rules have been configured to enforce the same NSRs.

In this example, a highly simplified topology was taken into account, and the advantage resulted in the allocation of one less firewall and the configuration of two fewer rules. However, in more intricate topologies involving more NSRs, the advantage can be substantially greater, resulting in significant resource savings.

6.2.2 Correction of Heuristics-related Code

The second intervention made on the heuristics-related code involved removing lines of code relevant to the solver. Since this version is based on the heuristics directly derived from the one based on the MaxSMT problem, there were some remnants left. Given that the focus of this thesis is to conduct performance and optimality tests at a generic level and not with special components like *NATs* and *Load Balancers*, the function related to forwarders, which are the components used in the tests to connect all the various parts of the topologies used in the test campaigns, has been corrected. However, this is an operation that must be carried out for all other functions as well if in future works related to other functions will also be involved.

Finally, in the *configure()* function of the heuristics, port handling was added

to the security policies, a functionality not initially planned. During the test campaign, in the initial phase, port intervals in the policies were not considered, so the wildcard symbol "*" was used to denote the entire port range. However, to make the policies more complex and thus better stress the framework's behavior, they were taken into account during a second phase.

6.3 Topologies

As stated in Chapter 3, one of the two inputs required by the VEREFOO framework is a network topology upon which Network Security Functions are to be implemented to meet a set of specified requirements, which constitutes the second input. Therefore, the first fundamental step before starting the test campaign is to determine which topologies to use for testing purposes.

To conduct the tests, three distinct topologies were employed to evaluate the framework's behavior under various conditions. They are described as follows:

- *VPNConfB*: This is a topology that had already been utilized for other purposes and was adapted to test the framework on networks involving the use of VPNs. Image 6.4 illustrates the structure.

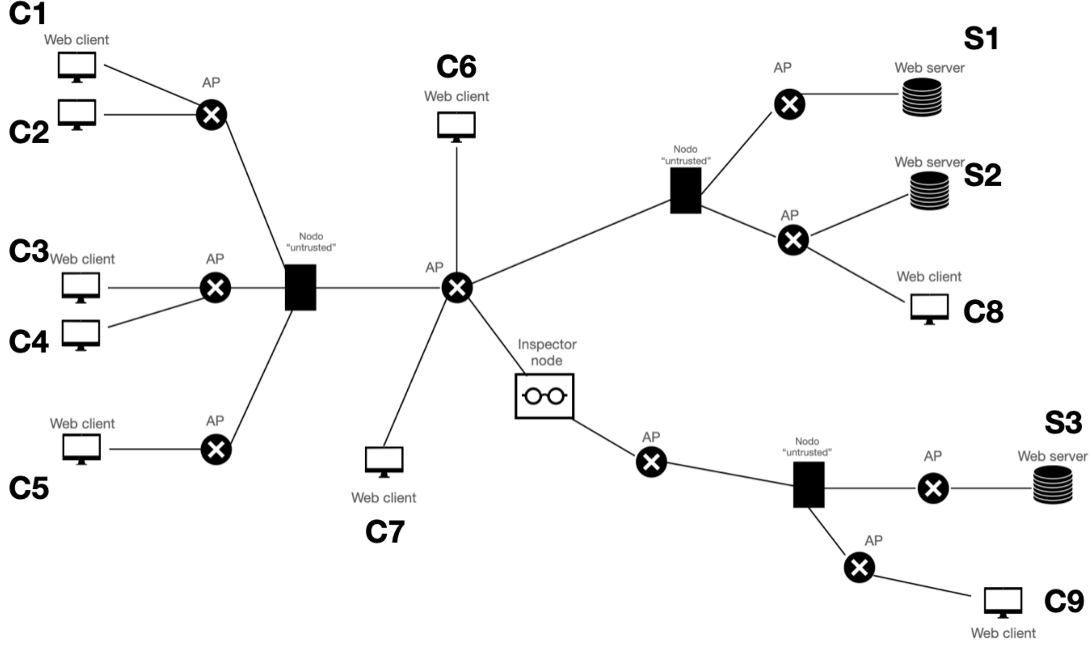


Figure 6.4: VPNConfB topology

The topology can be expanded and made more complex by adding additional endpoints on both the client and server sides, incorporating the basic structures depicted in Figure 6.5 that will be attached to the central AP.

Figure 6.5: VPNConfB topology basic structures

- *Geant*: this topology's name comes from Geant [17], which is an international organization that provides advanced computing and network resources for scientific research. The Geant network is a high-speed pan-European network connecting academic and research institutions in over 40 European countries, enabling them to share data, computational resources, and collaborate on research projects. Geant plays a crucial role in facilitating collaboration and knowledge exchange among researchers and institutions across Europe and beyond.

  The topology used by the Geant network is highly complex and branching, as depicted in Figure 6.6. However, the VEREFOO framework struggles to handle cases where there are multiple possible paths between 2 or more endpoints. Indeed, both algorithms for computing traffic flows, both Atomic Predicates and Maximal Flows described in Chapter 3, particularly the former which divides each flow into multiple sub-flows, must consider every single path present in the network. This inevitably leads to an exponential increase in total flows as the network size grows, quickly exhausting available memory.

  For this reason, the Geant topology has been adapted so that it can be evaluated with VEREFOO. The adapted topology is shown in Figure 6.7.

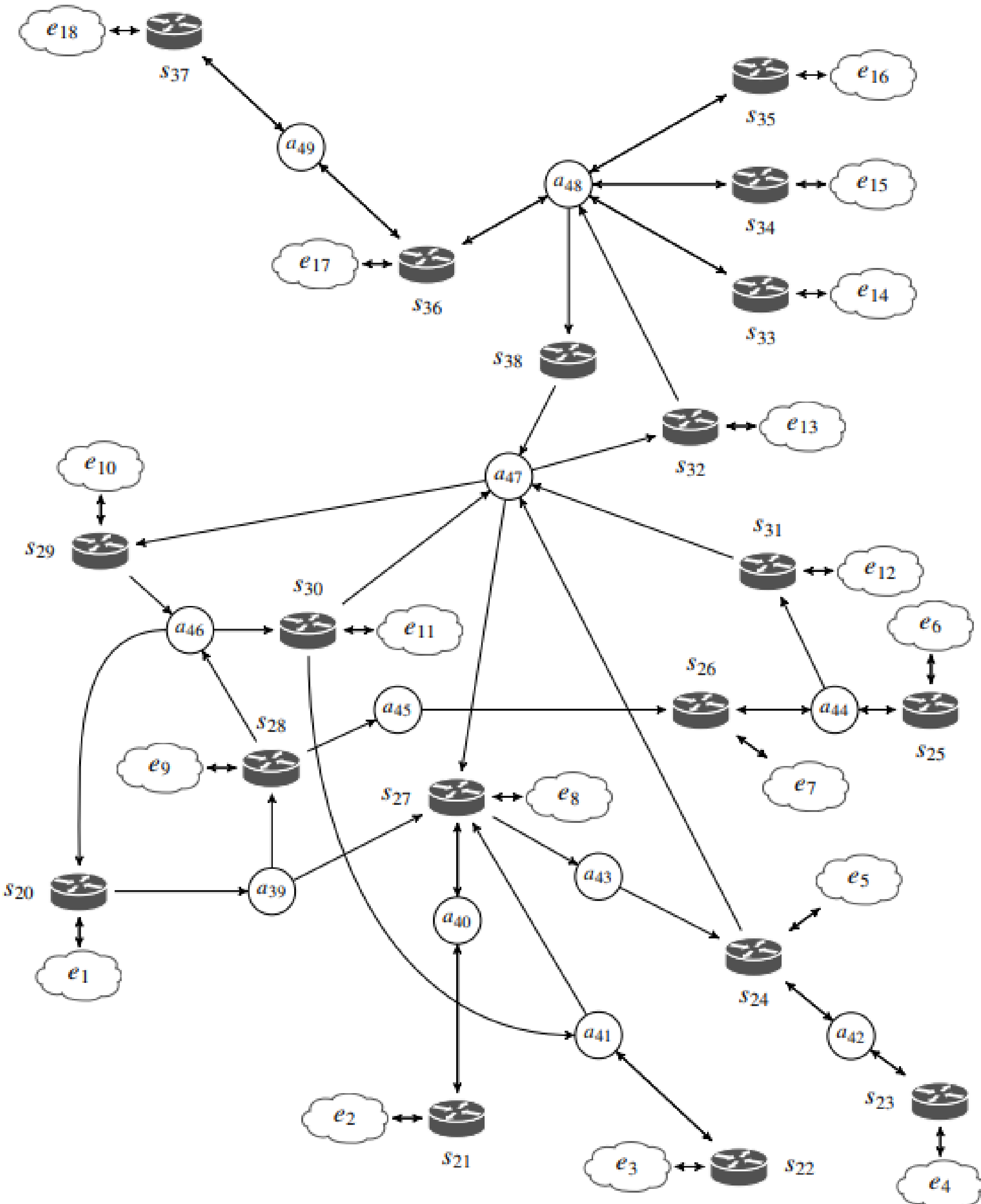


Figure 6.6: Allocation Graph inspired by Geant network topology

From Figure 6.7, it is apparent that the final topology consists of multiple basic structures, each of which is composed of a variable number of Web Clients, as determined by the user, connected to an AP via Forwarders. For each basic structure, which has a star shape, there are also two additional Forwarders connected to another AP, enabling traffic flow to and from other basic structures. This latter AP is ultimately connected to a central Forwarder that facilitates communication among all the substructures present in the topology.

When considering this topology, we can decide both the number of endpoints and the number of APs that will constitute the final topology. In particular,

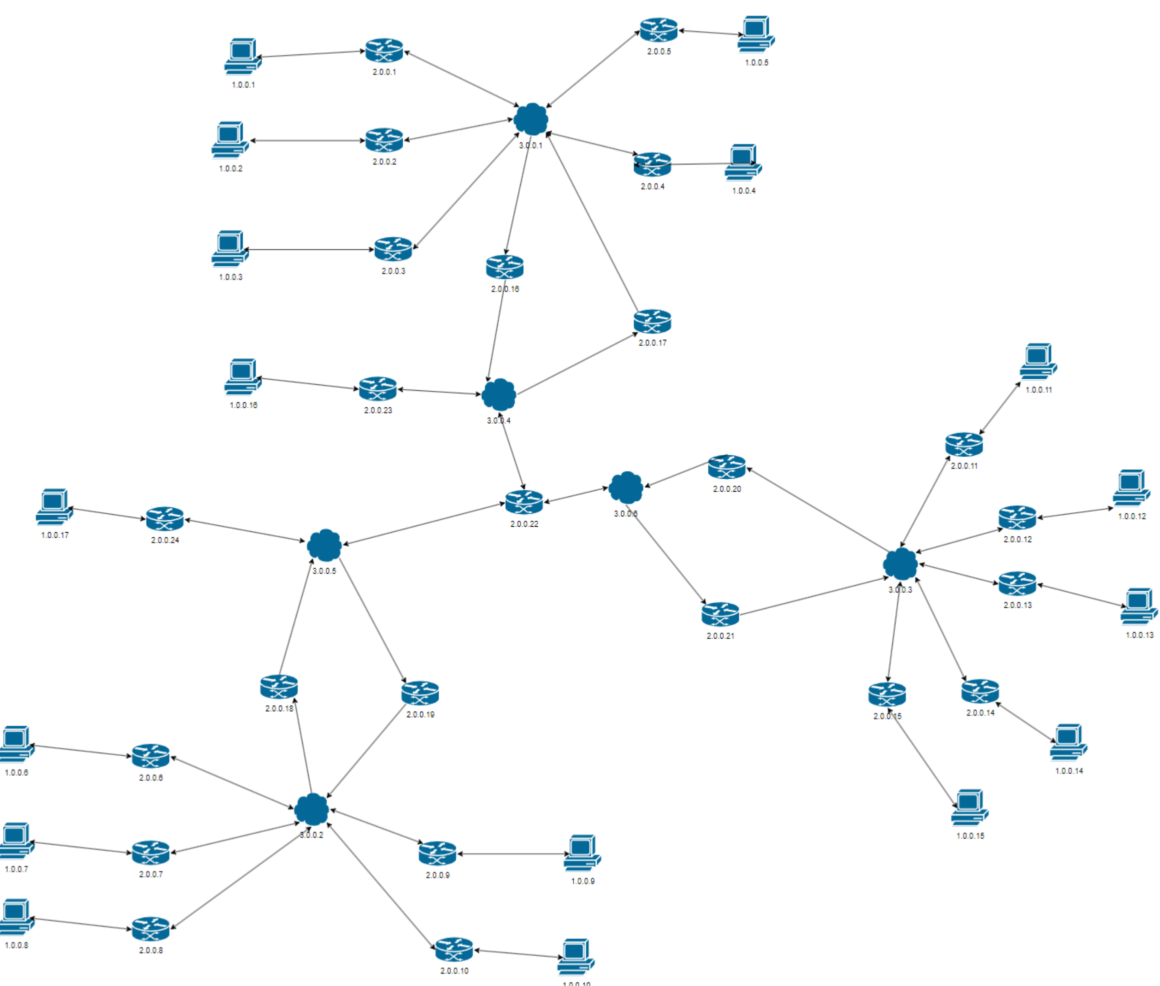


Figure 6.7: Example of Geant topology adapted to VEREFOO

we can decide the total number of Web Clients ($numberWebClients$) and the number of Web Clients for a single AP ($numberWebClientsPerAP$). The formula for computing the number of APs is the following:

$$numberAP = \left(\frac{numberWebClients}{numberWebClientsPerAP}\right) \times 2 \tag{6.1}$$

- *Internet2*: the name of this topology comes from Internet2 [18], which is a community of academic institutions, research organizations, industries, and government agencies in the United States collaborating to develop and implement advanced network technologies and Internet applications. Founded in 1996, Internet2 aims to promote innovation and advancement in networking and computer technologies to support scientific research, education, and the development of new Internet applications.

  Internet2 operates a high-performance research network, called the Internet2 Network, which provides higher bandwidth and performance compared to commercial Internet. This network is used to support a wide range of research and development projects in fields such as science, education, medicine, engineering, and more.

Internet2 network employs a less intricate network topology compared to Geant and is more linear in nature, as depicted in Figure 6.8.

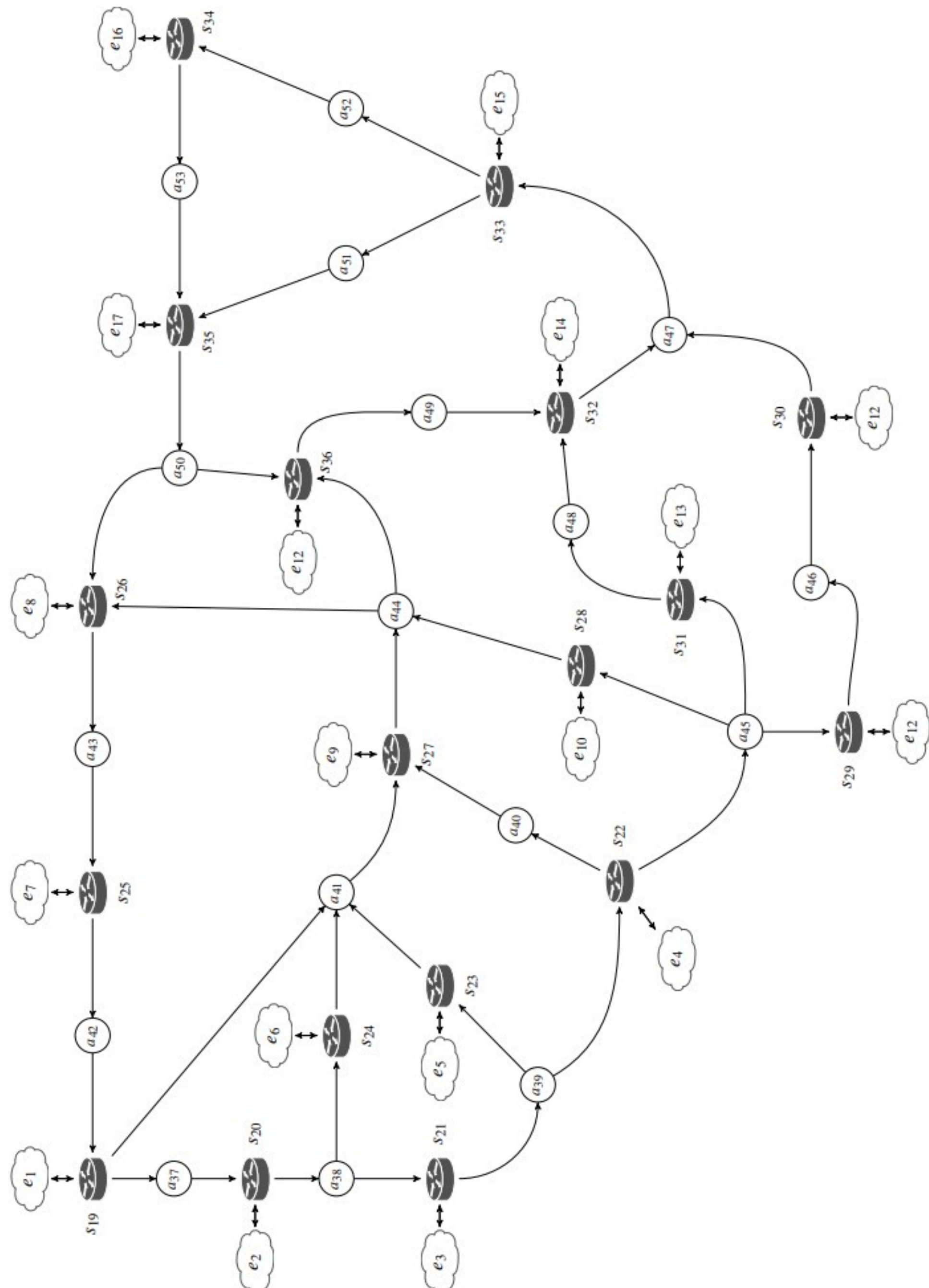


Figure 6.8: Allocation Graph inspired by Internet2 network topology

However, similar to the issue described for the Geant topology, this topology has also been adapted for execution in VEREFOO. An example is depicted in Figure 6.9.

This topology features base structures connected to a central forwarder, but with a different configuration compared to Geant. Unlike Geant's star-shaped topology, here the structures do not follow a star shape but rather consist of

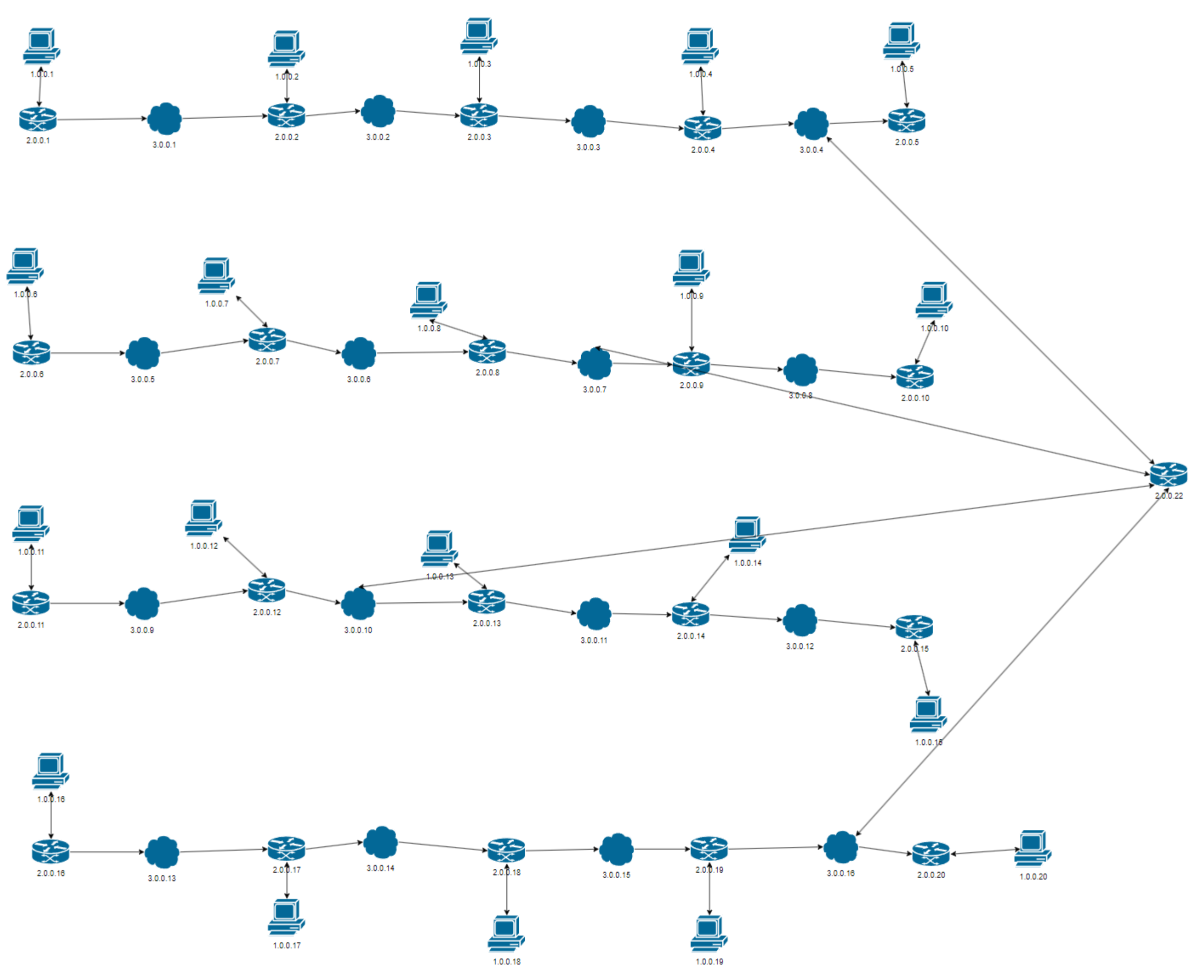


Figure 6.9: Example of Geant topology adapted to VEREFOO

chains of Web Clients connected to a Forwarder. Each Web Client is directly connected to a Forwarder, while Forwarders are interconnected via APs. This means that, while in Geant's topology the delivery of a packet destined to a Web Client is immediate once it reaches the central AP of the base structure, here the packet must traverse the entire chain of the base structure until it reaches the recipient, thus requiring multiple steps.

Even for this topology we can decide the number of APs of the final topology. In particular, we can decide the total number of Web Clients ($numberWebClients$) and the number of Web Clients per chain ($numberWebClientsPerChain$). The formula for the computation of the number of APs is the following:

$$numberAP = (numberWebClientsPerChain - 1) \times numChains \tag{6.2}$$

where $numChains$ is:

$$numChains = \frac{numberWebClients}{numberWebClientsPerChain} \tag{6.3}$$

6.4 Test Classes

In the context of the thesis, the Java classes created to execute tests on a framework can be described as entities that organize and manage code to perform specific operations or tests within the framework environment. These executable classes typically include a *main()* method serving as the entry point for program execution. Through the use of these classes, tests on the framework can be orchestrated and conducted, allowing for the evaluation of its performance, functionality, and behavior under various conditions.

Each topology to be tested is composed by two test classes:

- `TestPerformanceScalability`: This class handles all logistical aspects. For instance, it defines the total number of endpoints that will constitute the final topology, the number of APs if we want to feed the framework directly with an Allocation Graph, and the number of NSRs we want to create and implement on the final topology. During this work, a *test case* will be defined as an Allocation Graph with a set of NSRs that need to be implemented on it.

  Once all these parameters are defined, the associated test case is created through the described `TestCaseGenerator` class (which is described in more details below). This test case is then fed into the framework. Upon completion of the execution, this class also handles gathering results of interest, such as the *total execution time*, the *total number of configured firewalls*, and the *total number of configured rules*.

  In addition to managing these parameters, this class also allows for the selection of the test execution mode:

  - Creation of a single test case that will be processed only once by the framework.
  - Creation of a single test case that will be processed multiple times by the framework. This scenario can be useful when seeking to obtain a more precise and accurate result by averaging the outcomes.
  - Creation of multiple different test cases with incrementally increasing parameters. This scenario can be beneficial for testing the framework's behavior under varying complexities. For example, with each run of the framework, the number of endpoints in the network can increase by a certain number of units, or the number of NSRs, or a combination of both. For each test case, it can be decided whether to run the framework once or multiple times, as described in the previous cases.

  Finally, through this class, it is also possible to establish a termination condition for the tests. Examples can be:

  - The number of NSRs exceeds a certain threshold;
  - The number of endpoints exceeds a certain threshold;

  - The execution time exceeds a certain threshold;

  and so on.

- `TestCaseGenerator`: this class is invoked by the `TestPerformanceScalability` class, and is responsible for gathering parameters from it and using them to create the actual topology, along with the NSRs to be implemented.

  While the `TestPerformanceScalability` class is more or less the same for all topologies, this one varies depending on the specific topology, as each topology differs from the others.

  Here's an outline of the sequence of operations performed by a class of this type. Note that this is a general list and may vary slightly from one topology to another, depending on their specific structure.

  - Creation of all the endpoints (they can be Web Clients, Web Servers, other NFs or subnetworks);
  - Creation of Allocation Places if we are considering an Allocation Graph;
  - Creation of Forwarders that, based on the specific structure of each topology, connect endpoints and APs in a certain manner;
  - Creation of a set of NSRs taking into account the newly created endpoints. This point is better described in the next Section.

6.5 Network Security Requirements Generator

Given that the aim of this thesis is to test the performance of the various versions of the framework as the complexity of the topology and the number of endpoints vary, a custom Network Security Requirements generator has been developed. Specifically, this generator has the following characteristics:

- The generator offers the flexibility to select between two types of requirements: *isolation requirements* and *reachability requirements*. Additionally, users can specify the percentage of each type they desire. For instance, they may opt for exclusively isolation requirements, solely reachability requirements, or a blend of both types, such as 70% isolation and 30% reachability.

  This capability allows for fine-tuning the generated requirements according to specific testing needs and scenarios, enhancing the versatility of the testing process.

- For each generated requirement, both the selection of the source node and the destination node are randomly chosen. This deliberate randomness prevents the generator from creating overly specific policies, ensuring that the framework's behavior is tested under a variety of conditions.

  In real-world network scenarios, policies often need to accommodate endpoints located at various distances from each other within the network. Therefore,

by allowing random selection of nodes, the generator simulates a more realistic and diverse network environment, enabling comprehensive testing of the framework's performance across different configurations.

- The generator also provides the option to include or exclude port intervals when creating NSRs. In its simplest form, all policies use the wildcard symbol "*" to represent the entire port range for both source and destination, making the policy solely dependent on the source and destination IP addresses. But this approach may pose limitations for small-scale networks, as only a very limited number of policies can be created in this manner. This is because, for the framework to execute correctly, the policies must be conflict-free; otherwise, the result will be UNSAT.

  However, for more realistic and intricate policies, the inclusion of ports can be opted for. In this case, the generator randomly selects ports for both the source and destination. These ports can either be singular (e.g., 80), a range of ports (e.g., 100-200), or encompass the entire port range from 0 to 65535, denoted by the symbol "*".

  This added flexibility enables the generation of policies that better reflect real-world scenarios, allowing for more comprehensive testing of the framework's performance under various conditions.

Chapter 7

Test Campaign - First Phase

This chapter and the following ones represent the true core of this thesis work, as they present the results obtained from performance and scalability tests conducted using the tools and methods described in Chapter 6.

In particular, this Chapter compiles the results derived from an initial test campaign. The primary goal of this campaign was to understand the main differences between the four versions of the framework described in Chapter 6, namely:

- MaxSMT with Atomic Predicates
- MaxSMT with Maximal Flows
- heuristics with Atomic Predicates
- heuristics with Maximal Flows

Therefore, the objectives of this initial phase were essentially twofold:

1. Determine the differences in terms of execution times and optimality between the two main models used to model the framework: the approach using the MaxSMT problem and the heuristic approach.

   Specifically, we expected that the approach using the MaxSMT problem formulation would yield longer execution times compared to the heuristic-based approach. In fact, the MaxSMT problem is inherently complex, especially if the number of constraints is high or if it involves complex theories. Even the number of variables and constraints can significantly influence the time required to find a solution. Problems with a large number of variables or constraints, which in this case are represented by the number of endpoints and the number of NSRs, take more time to solve.

   However, we anticipated that the level of optimality, expressed in terms of the number of allocated firewalls and rules, would be higher. This translates to a lower total number of firewalls and rules compared to the heuristic approach.

2. Determine the differences in terms of execution times and optimality between the two approaches considered for modeling traffic flows: Atomic Predicates and Maximal Flows.

   As described in Chapter 4, concerning the approach based on MaxSMT, we anticipated that the solver would take more time using Maximal Flows because, as each Maximal Flow is not unique, it cannot be represented with a simple integer, which heavily impacts the solver's operation time. The algorithm for Atomic Predicates, on the other hand, takes more time to compute initially. However, once computed, the solver takes less time to handle them.

   Regarding the heuristic approach, however, we expect that Maximal Flows would perform better in terms of execution time because the algorithm for caomputing them is faster compared to Atomic Predicates, yet without the added complexity of the solver.

The tests were firstly conducted only on the topology *VPNConfB* considering two different scenarios:

1. **First scenario**: fixed number of endpoints and variable number of NSRs.
2. **Second scenario**: Variable number of endpoints and variable number of NSRs.

Regarding the requirements, initially, they do not include port considerations. Therefore, each requirement has the wildcard symbol "*" in both the *src_port* and *dst_port* fields. It means that each requirement is only based on the *source* IP address and on the *destination* IP address.

Two different cases were considered in each test to evaluate their effect on the overall performance of the framework:

1. 100% Isolation Requirements
2. 50% Isolation Requirements and 50% Reachability Requirements

The following parameters have been evaluated:

- Execution time
- Total number of allocated firewalls
- Total number of configured rules

7.1 First Scenario - Fixed Number of Endpoints and Variable Number of NSRs

In this scenario, a small network composed of a predefined number of endpoints was considered, structured as follows:

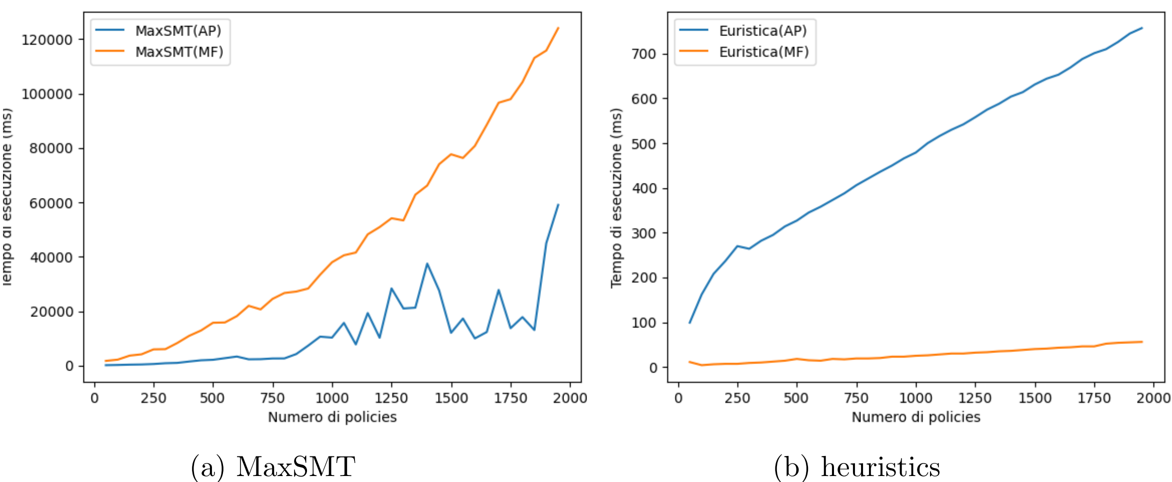
- Number of web clients = 30
- Number of web servers = 15
- Number of APs = 30
- Number of clients per AP = 2

At each iteration, the number of NSRs was increased. Due to the structure of the random requirement generator, the maximum limit for this network is 1980 = (number of endpoints) * (number of endpoints - 1). Below are the results presented through graphs.

7.1.1 100% Isolation Requirements

Execution Times

The first parameter considered during this testing phase is the execution time. Specifically, a maximum execution time of 30 minutes was set as the limit, which is equivalent to 1,800,000 milliseconds.



(a) MaxSMT (b) heuristics

Figure 7.1: Execution times

Figure 7.1a shows that, when using the MaxSMT problem, the execution times with Atomic Predicates is not linear but rather resembles an increasing exponential trend, despite marked fluctuations in the data. Overall, a significant growth

trend is observed. Similarly, concerning Maximal Flows, we observe a clear upward trend. Unlike the previous case, the trend is characterized by lower fluctuations but overall higher execution times, as we expected. Again, the growth appears to be exponential.

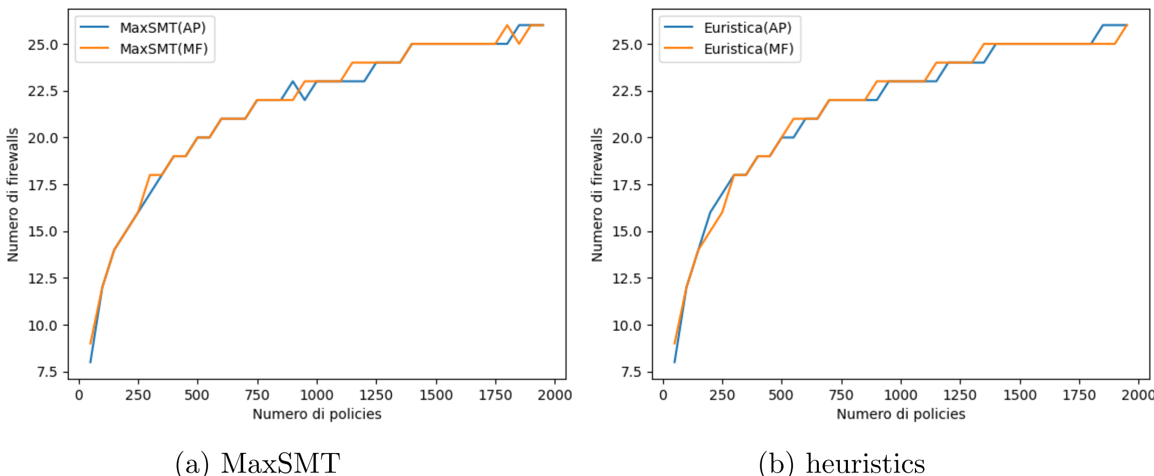
Figure 7.1b instead shows that, in the context of the heuristics,it is evident that the trends of the two versions no longer follow an exponential model as noted in the case of the solver. Instead, the trends appear to be much more linear, with reduced variability in execution times. In contrast to the previous case, and as we expected, the approach with Maximal Flows proves to be much more efficient compared to the one based on Atomic Predicates.

As can be seen from the graph concerning the heuristics, the execution time limit is well below half an hour, but it has simply been ”trimmed” to facilitate comparison between the various versions of the framework. More exhaustive tests on the execution time of the heuristics will be presented in the subsequent sections.

In general, we observe that the data related to the solver exhibit a significantly higher growth rate compared to those of the heuristics. Furthermore, the fluctuation in solver execution times is more pronounced, indicating a higher sensitivity to increasing the number of NSRs.

On the other hand, considering the heuristics, we note that for the same number of requirements, it is extremely faster compared to the solver. This gap is evident from the significant difference in the orders of magnitude of execution times. While the heuristics has execution times on the order of milliseconds or tenths of a second, the solver requires times on the order of seconds or even minutes, especially when using Maximal Flows.

Number of Allocated Firewalls



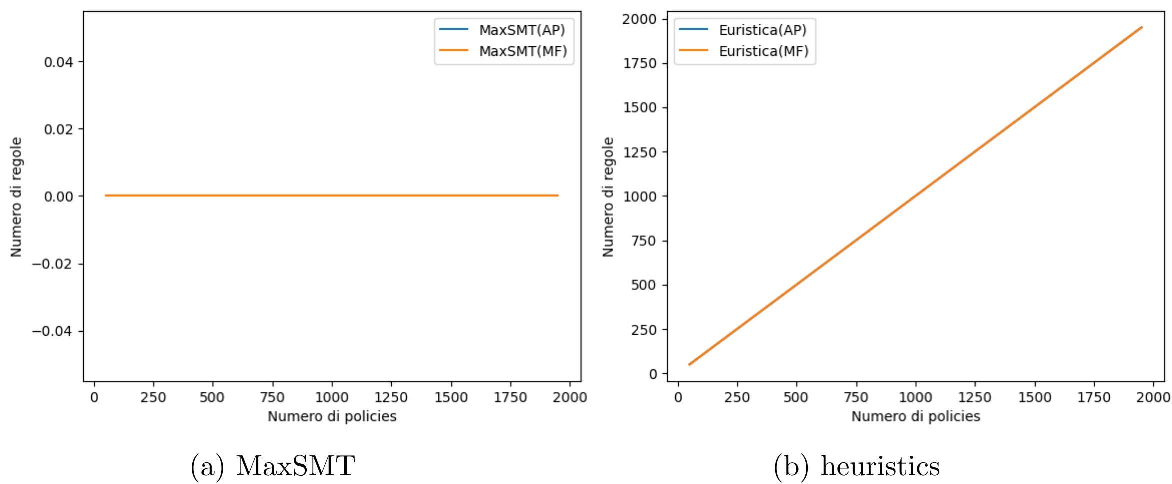
(a) MaxSMT (b) heuristics

Figure 7.2: Number of allocated Firewalls

As evident from Figures 7.2, the number of allocated firewalls is essentially the same for all four versions, given an equal number of endpoints and policies.

Therefore, in terms of the number of allocated firewalls, the heuristics exhibits an excellent level of optimality, comparable to that of the solver.

Number of Configured Rules



(a) MaxSMT (b) heuristics

Figure 7.3: Number of configured rules

From Figure 7.3, it can be noted that what differs significantly is instead the number of configuration rules applied on the firewalls.

In the case where there are 100% isolation requirements, if we consider the solver, a total of 0 rules is allocated, meaning that each firewall only has its default rule, which in this case is *DENY*. This indicates a simple and minimal configuration that satisfies the NSRs by configuring the least possible number of rules.

On the other hand, the heuristics adopts a different approach by allocating on the firewalls a total number of rules equal to the number of NSRs.

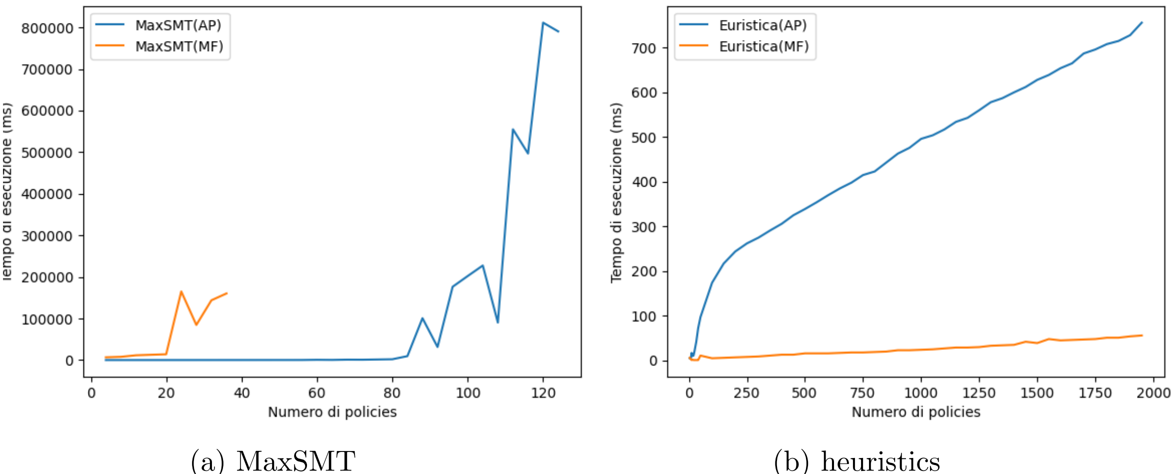
From this perspective, therefore, the heuristics appears to be less efficient than the solver in terms of rule configuration efficiency.

7.1.2 50% Isolation Requirements and 50% Reachability Requirements

Now the same tests that have just been presented will be repeated, but with a variation: instead of considering only isolation requirements, half isolation and half reachability requirements will be considered. This choice has been made to evaluate the framework's behavior following the addition of complexity due to reachability requirements, which puts more stress on the framework.

Execution Times

Observing Figure 7.4a, it is evident that in the version based on the MaxSMT problem, the execution times increase drastically, especially when the number of



(a) MaxSMT (b) heuristics

Figure 7.4: Execution times

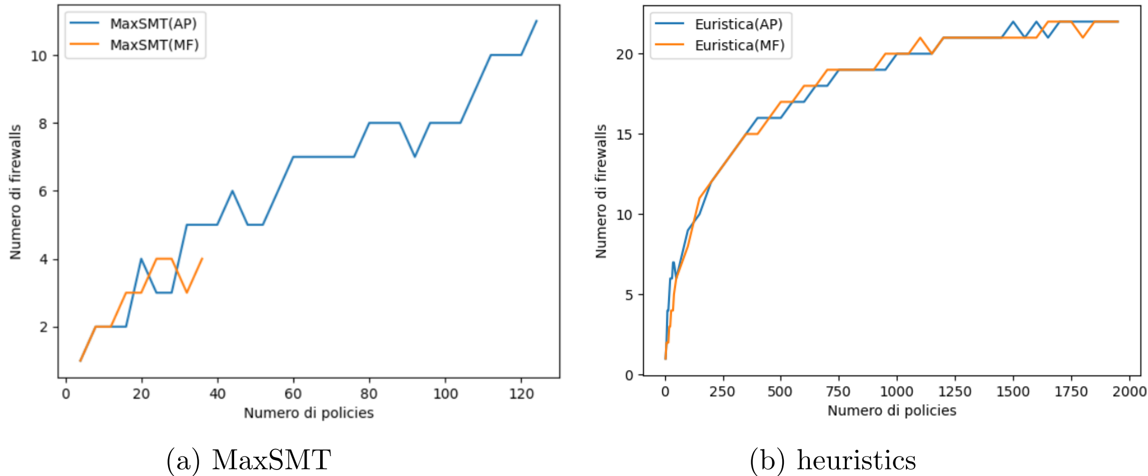
requirements exceeds a few tens of units, whereas in the previous case, it reached a few thousand. The reason for this significant difference lies in the complexity of firewall configuration, which is much higher when both reachability and isolation requirements are present. In particular, the version utilizing Maximal Flows begins to show a significant increase in execution times starting from around 40 NSRs, while the version with Atomic Predicates reaches similar thresholds at around 120 NSRs.

On the other hand, observing Figure 7.4b, concerning the heuristics, we can notice that there are no significant differences compared to the previous case where the entire set of requirements was focused on isolation. In other words, the heuristics maintains greater stability in execution times, regardless of the type of requirements.

In contexts where various types of requirements are present, the differences between MaxSMT-based and heuristics-based approaches become more pronounced and significant. Heuristics demonstrate remarkable adaptability, enabling efficient handling of scenarios with thousands of NSRs. Meanwhile, solvers exhibit clear limitations, as they can handle only a limited number of NSRs (typically just a few dozen). Additionally, even with such a limited number of NSRs, solvers show very high execution times, rendering them impractical for complex scenarios with a wide range of requirements.

Number of Allocated Firewalls

Observing Figure 7.6a, it can be noted that, concerning the MaxSMT problem, it is challenging to make a comparison between Atomic Predicates and Maximal Flows regarding the number of allocated firewalls because the approach based on Maximal Flows terminates "too soon" compared to Atomic Predicates. However, from the limited available data, it seems that, for the same network size and NSRs, even when mixed requirements are present, there are no significant differences.

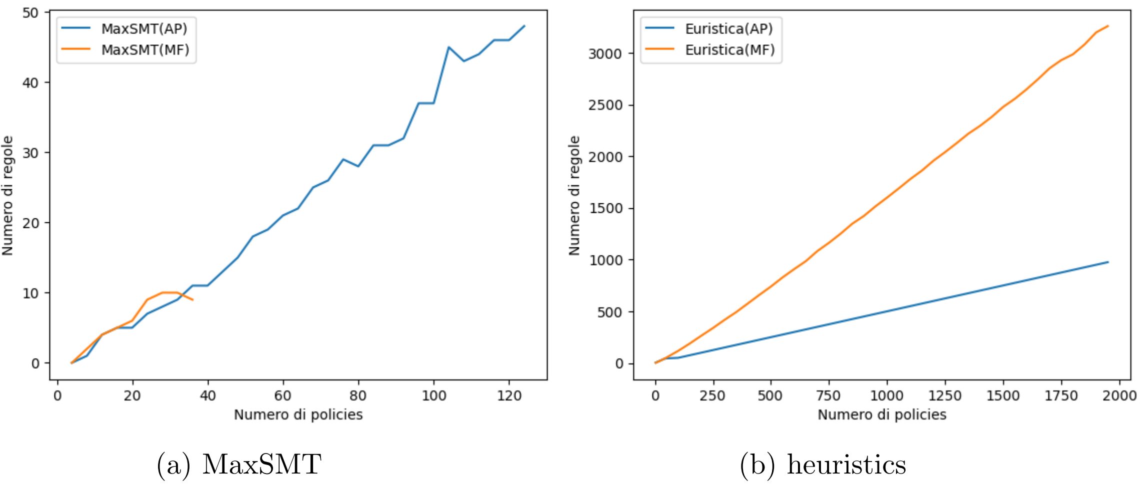


(a) MaxSMT (b) heuristics

Figure 7.5: Number of allocated Firewalls

Regarding the heuristics, however, from Figure 7.6b, it can be noticed that the number of allocated firewalls does not differ between Atomic Predicates and Maximal Flows, as in the case when there are 100% isolation requirements.

Number of Configured Rules



(a) MaxSMT (b) heuristics

Figure 7.6: Number of configured rules

The substantial difference, as in the previous case, lies in the number of configured rules. Considering the heuristics, thus Figure 7.6b, it is noticeable how the use of Atomic Predicates tends to allocate fewer rules compared to the use of Maximal Flows. In fact, focusing on Maximal Flows, it can be observed that more rules may be allocated than the number of requirements to be satisfied, which is quite unusual. During the execution of the tests, it was noted that this version of the framework, sometimes, tends to allocate the same rule on the same firewall. In particular, when there are multiple possible paths between the source and destination

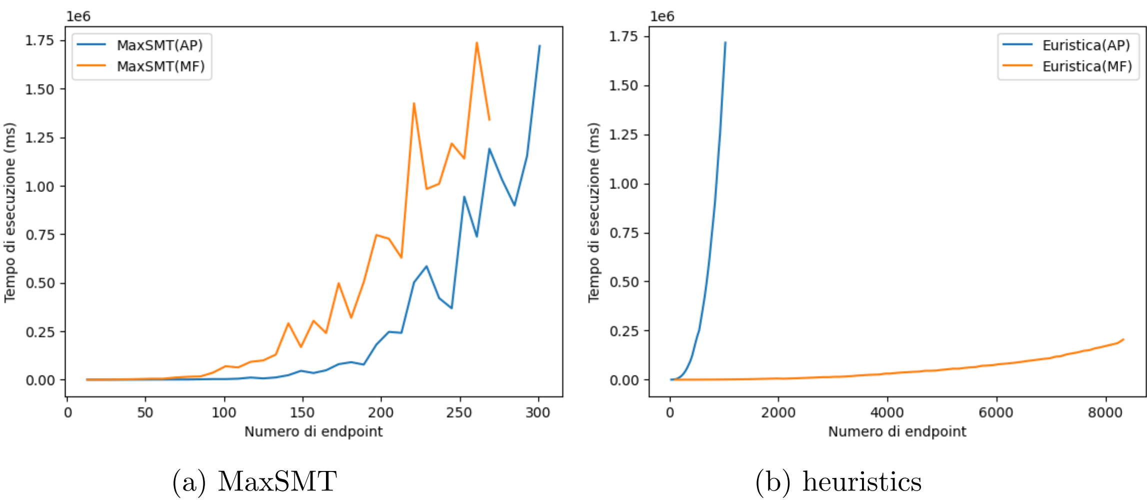
of a NSR, this results in multiple rules being allocated in the firewall. If there is only one path between the source and destination, a single rule is allocated in the firewall. However, if there are two or more paths, the same rule will be allocated multiple times in the firewall, one for each path.

7.2 Second Scenario - Variable Number of Endpoints and Variable Number of NSRs

In this second scenario, the goal is to test the behavior of the framework not only as the number of NSRs varies on a network of fixed dimensions but also on a network that grows in the number of endpoints. At each iteration, the number of endpoints in the network increases, and the ratio between the number of endpoints and the number of NSRs remains constant at 1:3. This implies that for a network with X endpoints, there will be 3X requirements, and so on.

7.2.1 100% Isolation Requirements

Execution Times



(a) MaxSMT (b) heuristics

Figure 7.7: Execution times

Observing Figure 7.7a related to the versions based on maxSMT, it's evident how the execution times increase exponentially with the growth of the network size and the number of NSRs, with significant fluctuations in the data. Both approaches, Atomic Predicates and Maximal Flows, show a very similar trend, but what emerges clearly is that the Maximal Flows approach consistently has higher execution times compared to the Atomic Predicates approach. This gap remains stable as the input data increases, unlike the previous scenario where the difference in execution times was not always stable.

For both approaches, the time limit is reached with a network composed of approximately 300 endpoints and 900 requirements.

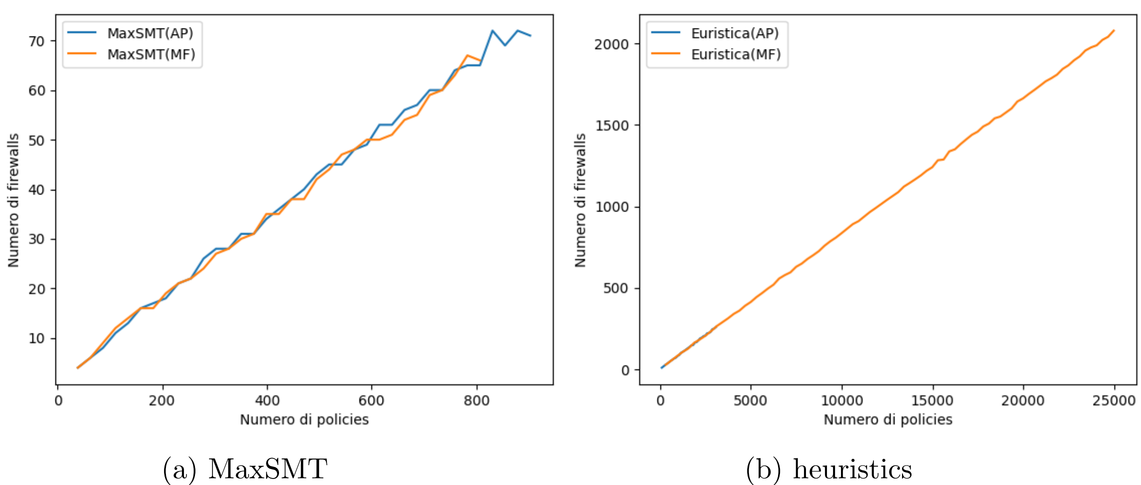
In the case of the heuristics, how it is shown in Figure 7.7b it's evident that the difference between the Atomic Predicates and Maximal Flows approaches is significantly more pronounced compared to the previous scenario. Specifically, the Atomic Predicates approach shows an exponential growth rate, while the Maximal Flows approach follows an exponential trend only when the network becomes significantly larger.

This implies that with the heuristics, the Atomic Predicates approach tends to reach the time limit with a network of around 1000 endpoints and 3000 requirements due to its rapid increase in execution times, while the Maximal Flows approach manages to handle larger problem sizes before exhibiting an exponential growth rate in execution times.

In general, the heuristics-based approach demonstrates remarkable scalability, being able to handle networks composed of thousands of endpoints and a high number of NSRs. This means it can effectively and efficiently tackle large-scale scenarios.

On the other hand, the MaxSMT-based approach shows significantly lower scalability, being capable of handling only networks with a relatively limited number of endpoints (a few hundred) and consequently, requirements before reaching the time limit.

Number of Allocated Firewalls

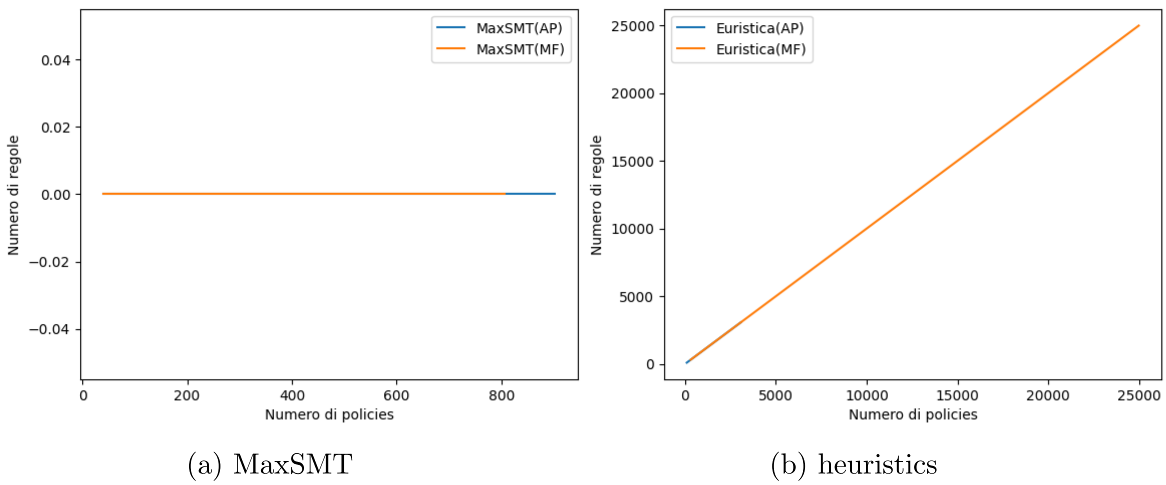


(a) MaxSMT (b) heuristics

Figure 7.8: Number of allocated Firewalls

From Figure 7.9, it's evident that the number of firewalls allocated by both the solver and the heuristics, given an equal number of endpoints and requirements, is essentially the same, as in the previous scenario.

Number of Configured Rules



(a) MaxSMT (b) heuristics

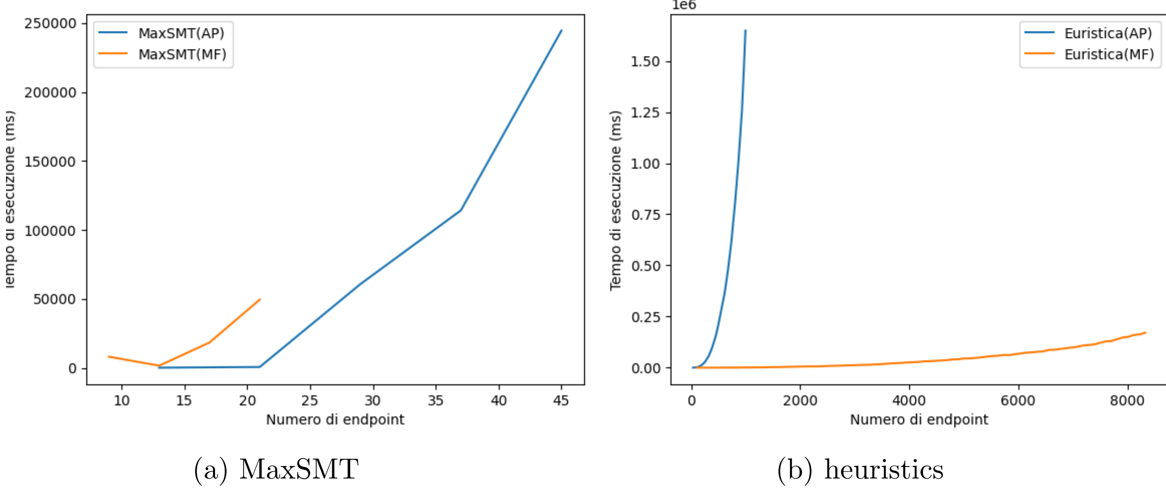
Figure 7.9: Number of configured rules

What significantly differs is the number of rules configured.

Indeed, as in the previous scenario, in the case of the solver, 0 rules are allocated (meaning that each firewall has only the default rule), while the heuristics allocates a number of rules equal to the number of requirements, and the same considerations as before apply even in this case.

7.2.2 50% Isolation Requirements and 50% Reachability Requirements

Execution Times



(a) MaxSMT (b) heuristics

Figure 7.10: Execution times

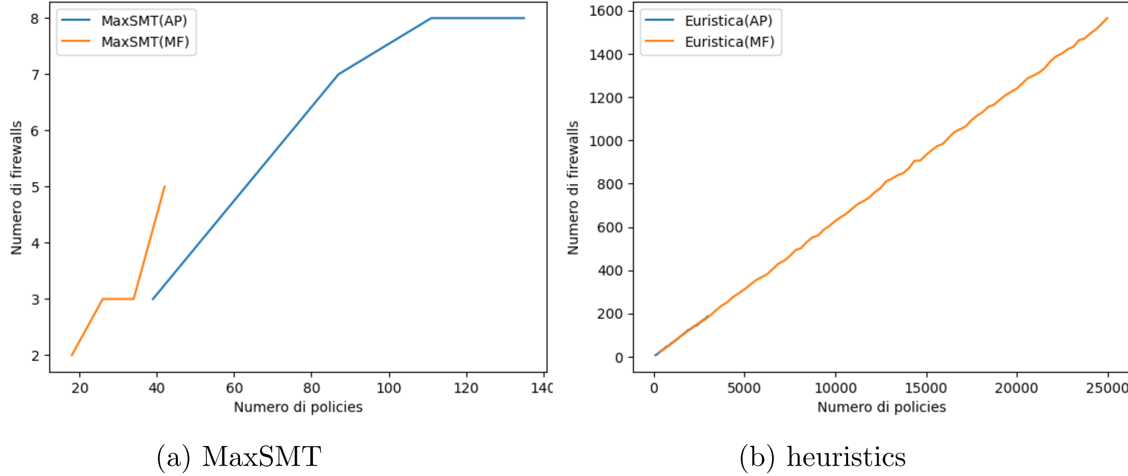
In this scenario, when concerning the Max-SMT approach (Figure 7.10a) we observe that the execution times increase significantly and follow an exponential trend, especially when the network size exceeds a few tens of endpoints. Similar to what we've seen before, when dealing with a combination of different types of NSRs, the time limit is reached much earlier due to the increased complexity required in configuring the firewalls.

Specifically, the version usingg Maximal Flows reaches the time limit with a network of approximately 20 endpoints and 60 NSRs, while the version with Atomic Predicates exhibits a similar behavior, reaching the limits with a network of about 45 endpoints and 135 NSRs.

Regarding the heuristics (Figure 7.10b), we can observe that there are no significant differences compared to the previous scenario where the entire set of NSRs was isolation-oriented. In other words, even in this scenario, the heuristics maintains greater stability in execution times, regardless of the type of NSRs.

In the context where various types of requirements are present, the differences between the solver-based approach and the heuristics-based approach become more evident and significant, even compared to the previous scenario. The heuristics demonstrates remarkable adaptability, allowing it to efficiently handle scenarios with thousands of endpoints and requirements, especially when using Maximal Flows. The solver shows clear limitations, as it can only handle small networks (a few tens of endpoints) and a limited number of requirements.

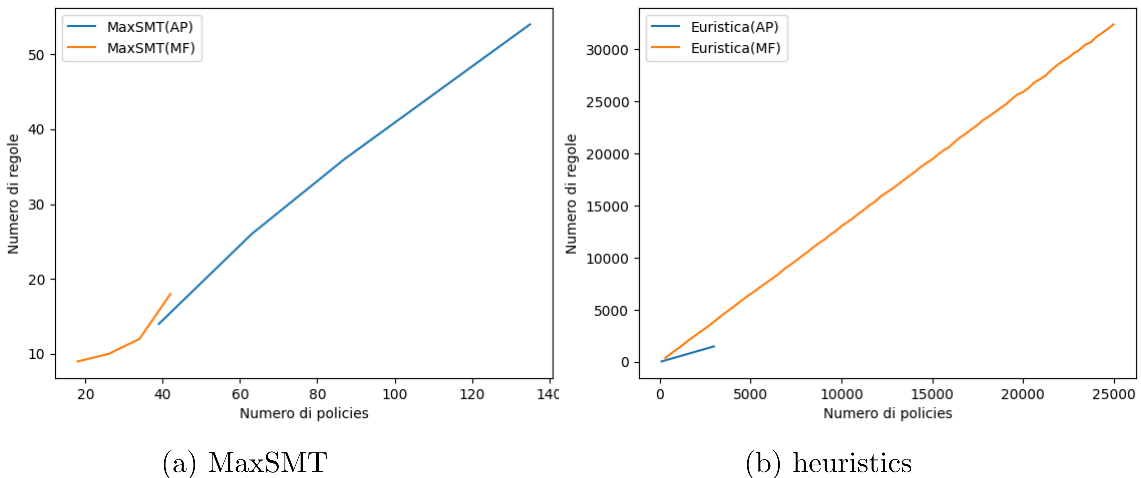
Number of Allocated Firewalls



(a) MaxSMT

(b) heuristics

Figure 7.11: Number of allocated Firewalls



(a) MaxSMT (b) heuristics

Figure 7.12: Number of configured rules

Number of Configured Rules

Observing Figures 7.11 and 7.12, there are no particular differences compared to the scenario where the set consisted solely of isolation requirements. Therefore, the same considerations made previously apply.

7.3 Considerations

The execution of the tests presented in this chapter has brought to light the following results:

- *Execution times*: the heuristics is generally faster than the Z3 solver for both Atomic Predicates and Maximal Flows.
- *Number of allocated firewalls*: both the heuristics and the solver allocate a similar number of firewalls at an equal network size and set of NSRs.
- *Number of configured rules*: the solver tends to allocate a significantly lower number of rules compared to the heuristics, both when there are only isolation requirements and mixed requirements. This results in a more minimal and simplified configuration of the firewalls.
- *Atomic Predicates vs Maximal Flows*: using MFs results in increased resolution times compared to APs when using the solver, while the heuristics shows a drastic decrease in execution times when switching from APs to MFs. This can be explained by the fact that if the solver is used, the majority of the execution time is occupied by solving the MaxSMT problem, while the calculation of MFs and APs, in comparison, is negligible. Additionally, the solver takes much more time if MFs is used as input compared to APs, as explained in Chapter 4. In contrast, the heuristics employs a non-exact but

significantly faster algorithm compared to the time it takes for the solver to solve the MaxSMT problem. In this case, the majority of the execution time is spent on calculating APs or MFs, while the time spent on the actual algorithm is negligible in comparison. This results in a much faster execution time using MFs compared to using APs, because computing MFs is much quicker.

- *Impact of mixed requirements*: the use of both isolation and reachability requirements introduces greater computational complexity when using the Z3 solver, but there are no significant differences when using the heuristics, demonstrating greater adaptability compared to the solver. The heuristics are less affected by the combination of different types of NSRs compared to the solver. This is evident in execution times, as the Z3 solver shows a drastic increase in times when mixed requirements are present, while the heuristics maintain more consistent execution times.

Chapter 8

Optimizing firewall Configuration for Minimal Rule Allocation

The results of the performance and scalability tests in Chapter 7 highlight a significant disparity between the heuristics and the solver. While the heuristics stands out for superior performance in terms of execution time, the solver and the heuristics exhibit similar behavior regarding the number of allocated firewalls. However, the main discrepancy lies in the number of configured rules. Specifically, when the entire set of NSRs consists of isolation requirements, the heuristics tends to allocate one rule for each requirement, while the solver utilizes a default *DENY* rule for each firewall in order to configure 0 rules on each of them. Even in the case of mixed requirements, the solver demonstrates a higher level of optimality, thanks to the use of firewall's default rules (in this case *DENY* or *ALLOW*), compared to the heuristics.

Additionally, another critical issue of the heuristics emerged during the tests conducted in Chapter 7: when there are multiple possible paths between the source node and the destination node of a requirement, the same rule is assigned multiple times to the same firewall, resulting in ineffective rule duplication.

This chapter aims to address these heuristics algorithm limitations, presenting a series of improvements aimed at increasing the level of optimality in the number of allocated rules and mitigating the inefficiency resulting from rule duplication. By implementing new heuristics strategies and optimizing rule allocation procedures, the goal is to achieve a more efficient firewall configuration that complies with defined security requirements.

8.1 Eliminating Redundant Rule Configuration

The first intervention involved eliminating the duplication of rules across firewalls. This enhancement was implemented within the *configure()* method of the heuristics, as described in Chapter 3. Before detailing the modification, it's essential to explain what an object of type *Elements* is in VEREFOO.

8.1.1 Elements class

In VEREFOO, the *Elements* class is a class that models a rule configured on a firewall object. It consists of the following properties:

- **action**: action of the rule (DENY or ALLOW);
- **source**: source IP address of the rule;
- **destination**: destination IP address of the rule;
- **protocol**: layer 4 protocol of the packet class to which the rule applies;
- **srcPort**: port or port range of the source node;
- **dstPort**: port or port range of the destination node.

Through the process described in Chapter 5.1, specific rules are configured for each requirement on every firewall allocated by the heuristics. Specifically, if it's an isolation requirement, a *DENY* rule is configured, whereas for a reachability requirement, an *ALLOW* rule is configured. The intervention implemented involves checking, before allocating a new rule to a firewall, whether an identical rule already exists on that firewall. If not, the new rule is allocated; otherwise, it is skipped. Listing 8.1 shows the code for an isolation requirement, but it is the same for a reachability requirement.

```
if(srType == PName.ISOLATION_PROPERTY) {
    Elements element = new Elements();
    element.setAction(ActionTypes.DENY);
    element.setSource(IPSS);
    element.setDestination(IPDS);
    element.setSrcPort(pSS);
    element.setDstPort(pDS);
    Boolean flag = false;

    for (Elements e : fw.getElements()) {
        if (isEqual(element, e) ) {
      flag = true;
   }
    }
    if (flag == false)
    fw.getElements().add(element);
}
```

Listing 8.1: Check whether a rule already exists on the firewall or not.

The *isEqual(Elements e1, Elements e2)* method is shown in Listings 8.2.

```
public boolean isEqual(Elements e1, Elements e2) {
    if (e1.getAction() == e2.getAction() &&
        e1.getSource().equals(e2.getSource()) &&
   e1.getDestination().equals(e2.getDestination()) &&
       e1.getSrcPort().equals(e2.getSrcPort()) &&
   e1.getDstPort().equals(e2.getDstPort())) {
```

```
    return true;
     }
     else {
    return false;
     }
}
```

Listing 8.2: Check whether a rule already exists on the firewall or not.

8.2 Post Processing of Rules

Another improvement made for each firewall allocated and configured by the heuristics was to replace one of the two sets of rules (all DENY rules or all ALLOW rules) with a default rule, thereby saving the configuration of a large number of rules. Below is the approach which has been used.

Let's denote:

1. A_A as the set of all Allocation Places in the topology;
2. $\mathbb{F}_a^D$ is the set of the denying rules deriving from isolation requirements;
3. $\mathbb{F}_a^A$ is the set of the allowing rules deriving from complete and partial reachability requirements.

For each $a \in A_A$ such that $allocated(a) = true$ (it means that a firewall is allocated in that a) the cardinalities of $\mathbb{F}_a^D$ and $\mathbb{F}_a^A$ are compared:

1. if $\mathbb{F}_a^D \geq \mathbb{F}_a^A$, then the firewall in a is configured with $DENY$ as default action and the filtering rules of the $\mathbb{F}_a^A$ set;
2. if $\mathbb{F}_a^D < \mathbb{F}_a^A$, then the firewall in a is configured with $ALLOW$ as default action and the filtering rules of the $\mathbb{F}_a^D$ set.

The implementation in shown in Listing 8.3.

```
public static void postProcessDisjunctRules(List<Node> nodes) {

      for (Node n: nodes) {

         if (n.getFunctionalType() == FunctionalTypes.firewall) {

         List<Elements> fRules =
             n.getConfiguration().getfirewall().getElements();
         List <Elements> dRules = fRules.stream().filter(r -> r.getAction()
             == ActionTypes.DENY).collect(Collectors.toList());
         List <Elements> aRules = fRules.stream().filter(r -> r.getAction()
             == ActionTypes.ALLOW).collect(Collectors.toList());

         Configuration confF = new Configuration();
         confF.setName("confF");
```

```
        n.setConfiguration(confF);
        firewall fw = new firewall();

        if(aRules.size() > 0 && dRules.size() > 0) {

          if(dRules.size() >= aRules.size()) {

            fw.setDefaultAction(ActionTypes.DENY);
            for(Elements e : aRules) {
              e.setProtocol(L4ProtocolTypes.ANY);
              fw.getElements().add(e);
                }
            }
           else if (dRules.size() < aRules.size()) {

                fw.setDefaultAction(ActionTypes.ALLOW);
              for(Elements e : dRules) {

                e.setProtocol(L4ProtocolTypes.ANY);
                fw.getElements().add(e);

                }
                }
            }
          else if (aRules.size() == 0 && dRules.size() > 0 ) {
            fw.setDefaultAction(ActionTypes.DENY);
            }
          else if (aRules.size() > 0 && dRules.size() == 0 ) {

            fw.setDefaultAction(ActionTypes.ALLOW);

            }
          confF.setfirewall(fw);
          n.getConfiguration().setfirewall(fw);


          }
        }

    }
```

Listing 8.3: Algorithm for post processing of rules.

8.3 Check with Verigraph

To verify that the algorithm for post-processing rules works correctly, it is necessary to test the produced output. Verigraph, a tool developed by the Netgroup working group at Politecnico di Torino, is used for this purpose. Verigraph exploits Z3 to define a formal model of a network, allowing for formal verification of the topology provided as input. In our case, the process involves:

- running the heuristics on a specific topology to enforce a set of NSRs;

- executing the post-processing algorithm described in section 8.2;
- taking the resulting XML schema and passing it as input to Verigraph;
- Verigraph will output "SAT" if the Network Security Functions present in the XML topology are good and sufficient to satisfy the defined NSRs, and "UNSAT" otherwise.

Chapter 9

Test Campaign - Second Phase

This chapter aims to conduct performance and scalability tests, similar to Chapter 7, while also evaluating the enhancements introduced in Chapter 8. The primary objective is to assess the degree of improvement in heuristics optimality resulting from these changes, particularly focusing on the number of configured rules. The tests were carried out based on the following criteria:

- Only two versions of the framework are considered in this chapter:
  - MaxSMT with Atomic Predicates
  - Heuristics with Maximal Flows

  This selection was made based on the findings from Chapter 7, indicating that the other two versions are less effective in terms of execution time.

- The Chapter includes testing on the two new topologies introduced in Chapter 6, namely the *Geant* topology and the *Internet2* topology, to analyze the framework's behavior across diverse topologies. For the *Geant* topology, the variable $numberWebClientsPerAp$ was set to 2 for the MaxSMT and to 5 for the Heuristics (but to 2 when comparing it to the MaxSMT). Instead, for the *Internet2* topology the variable $numberWebClientsPerChain$ was set to 2 for the MaxSMT and to 10 for the heuristics (but to 2 when comparing it to the MaxSMT). The formulas for computing the number of APs for both topologies was explained in Chapter 6.

- The tests specifically focus on scenarios where both the number of endpoints and requirements increase proportionally at each iteration, maintaining a 1:3 ratio between endpoints and requirements. The time limit for these tests remains set at 30 minutes (1,800,000 ms).

- Similar to Chapter 7, two test cases were considered:
  - 100% isolation requirements
  - 50% isolation requirements and 50% reachability requirements

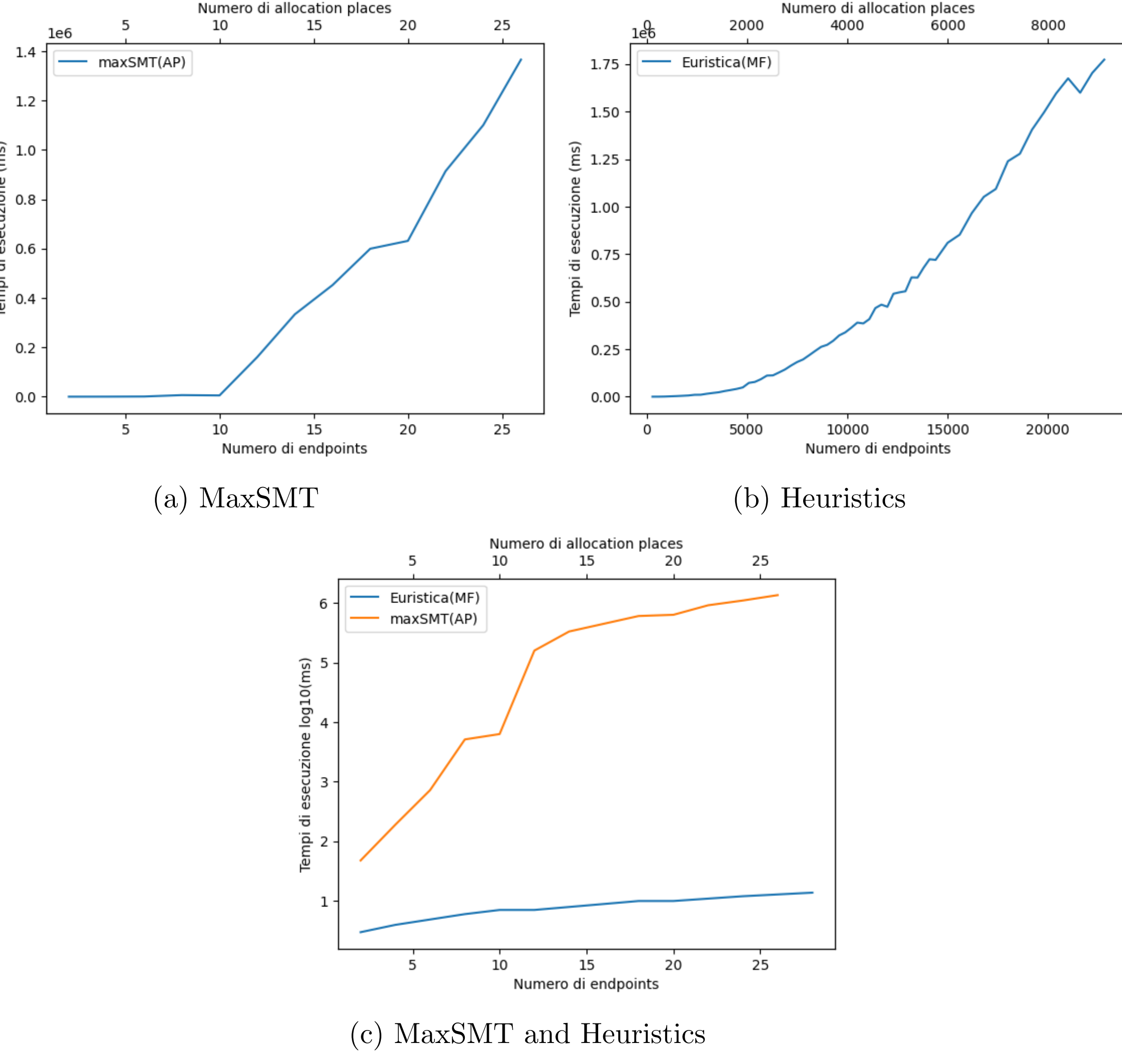
> The evaluation criteria for these tests include execution time, total number of allocated firewalls, total number of configured rules and total number of rules configured after Post Processing (only for heuristics).

Next, the results will be presented through graphs divided into sections. We first analyze execution times, comparing the solver and heuristic approaches. Then, we shift the focus to the number of allocated firewalls and examine the configured rules, completing the analysis.

9.1 100% Isolation Requirements

9.1.1 Execution Times

Geant Topology



(a) MaxSMT

(b) Heuristics

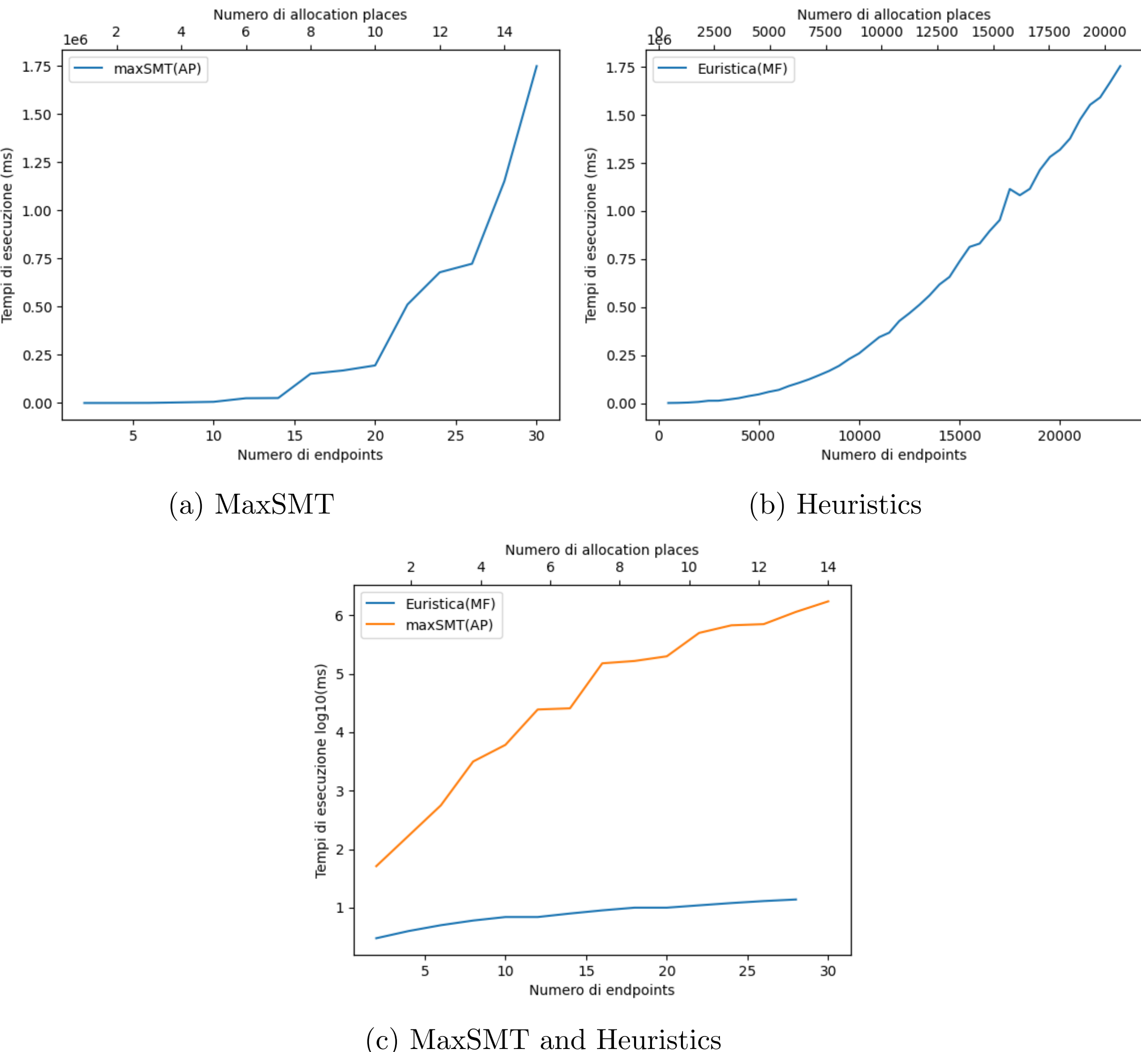
(c) MaxSMT and Heuristics

Figure 9.1: Execution times comparison - GEANT

The solver's data shows a higher growth rate compared to the heuristics, reaching time limits with a small network of 30 endpoints and about 90 NSRs. Up to 15 endpoints, the growth rate is quite linear. Beyond 15, it takes on an exponential connotation. On the other hand, considering the heuristics, we notice that it is faster compared to the solver. In fact, it reaches the time limits with a much larger network, consisting of approximately 25,000 endpoints and 75,000 NSRs. The differences in orders of magnitude are as follows:

- Up to 6 endpoints: 2 orders of magnitude;
- From 7 to 10 endpoints: 3 orders of magnitude;
- From 11 to 22 endpoints: 4 orders of magnitude;
- From 22 endpoints onwards: more than 5 orders of magnitude.

Internet2 Topology



(a) MaxSMT

(b) Heuristics

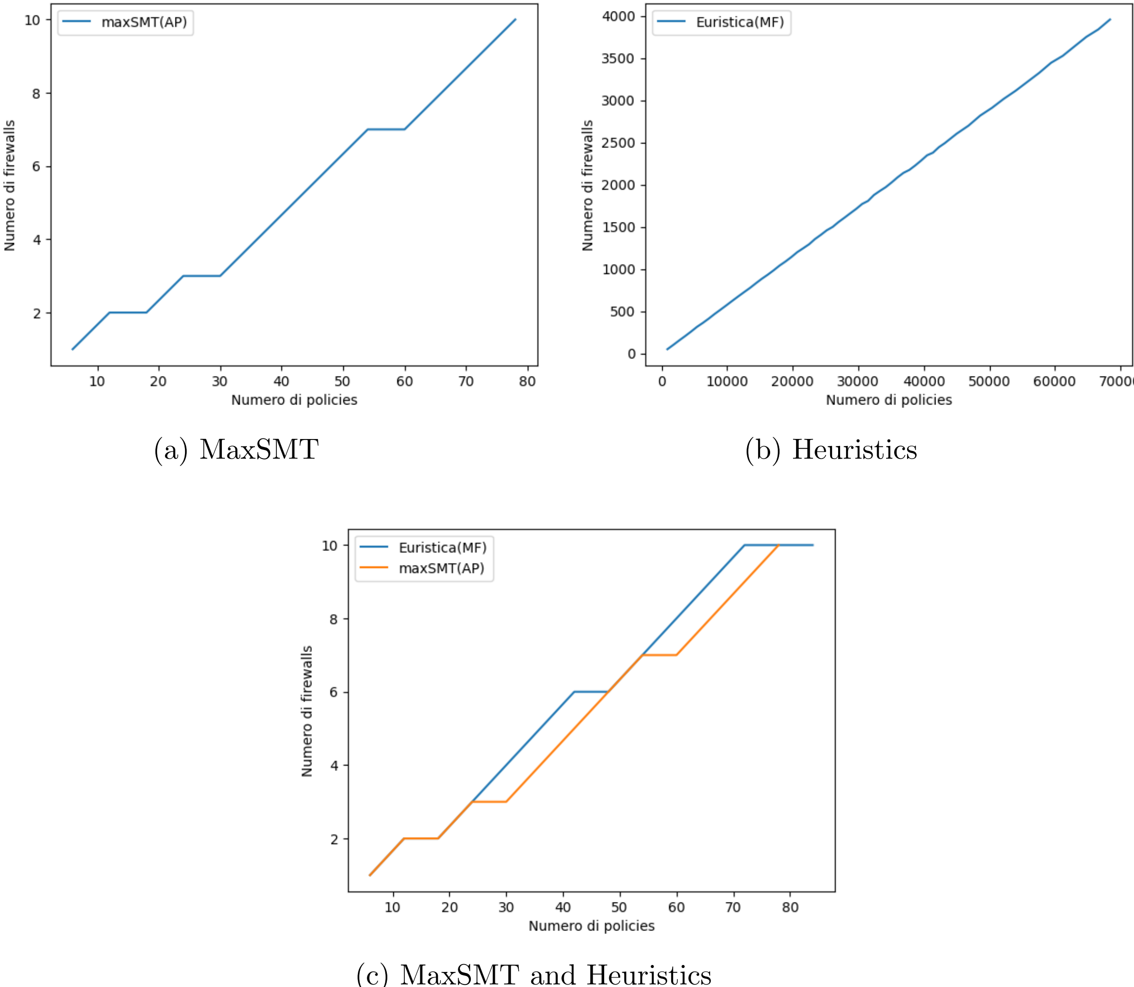
(c) MaxSMT and Heuristics

Figure 9.2: Execution times comparison - INTERNET2

The solver exhibits a higher growth rate compared to the heuristics in this topology too. It reaches time limits with a slightly larger network of about 30 endpoints and 90 requirements. Conversely, the heuristics handle a significantly larger network of around 25,000 endpoints and 75,000 requirements, maintaining faster execution times. The differences in orders of magnitude are the same as the *Geant* topology.

9.1.2 Number of Allocated Firewalls

Geant Topology

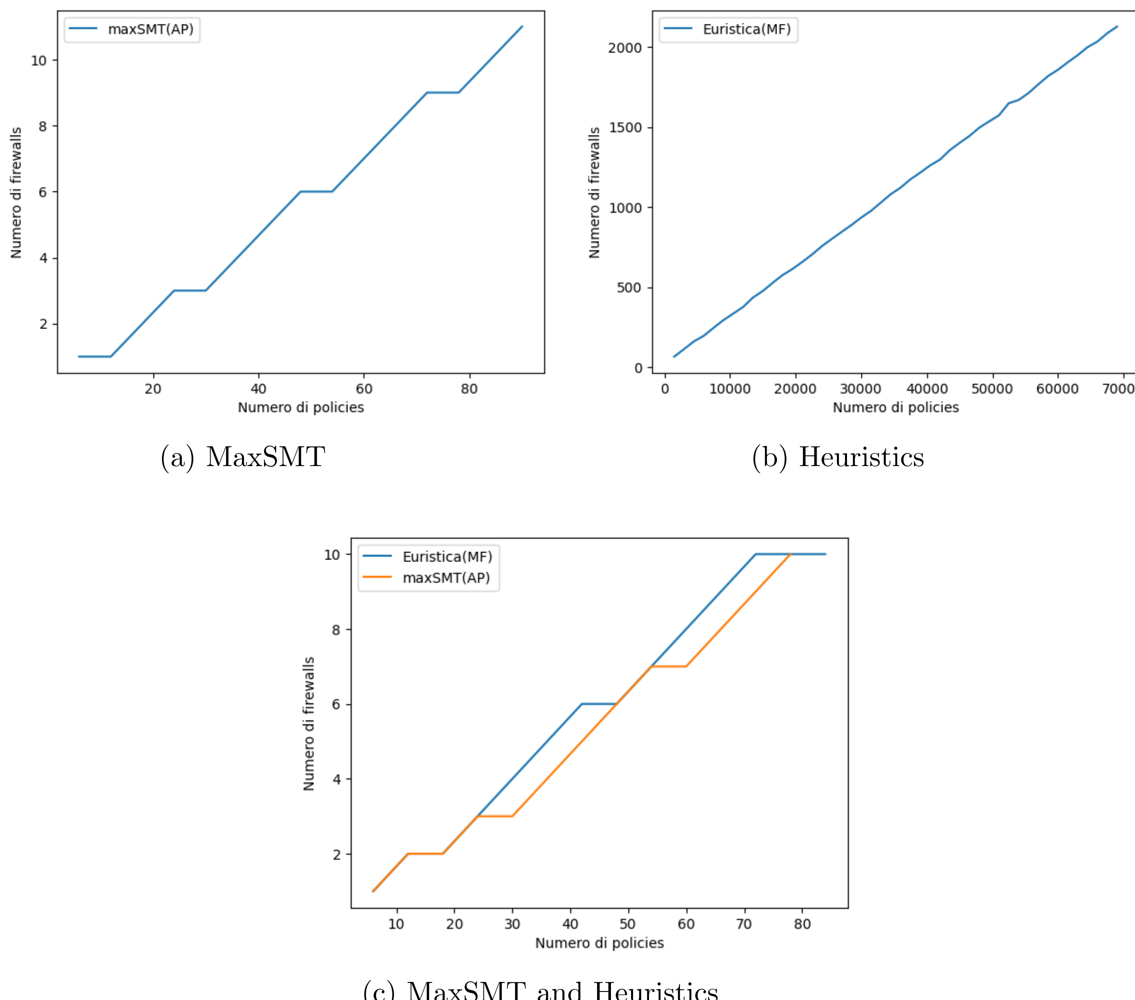


(a) MaxSMT (b) Heuristics

(c) MaxSMT and Heuristics

Figure 9.3: Number of allocated firewalls - GEANT

Internet2 Topology



(a) MaxSMT

(b) Heuristics

(c) MaxSMT and Heuristics

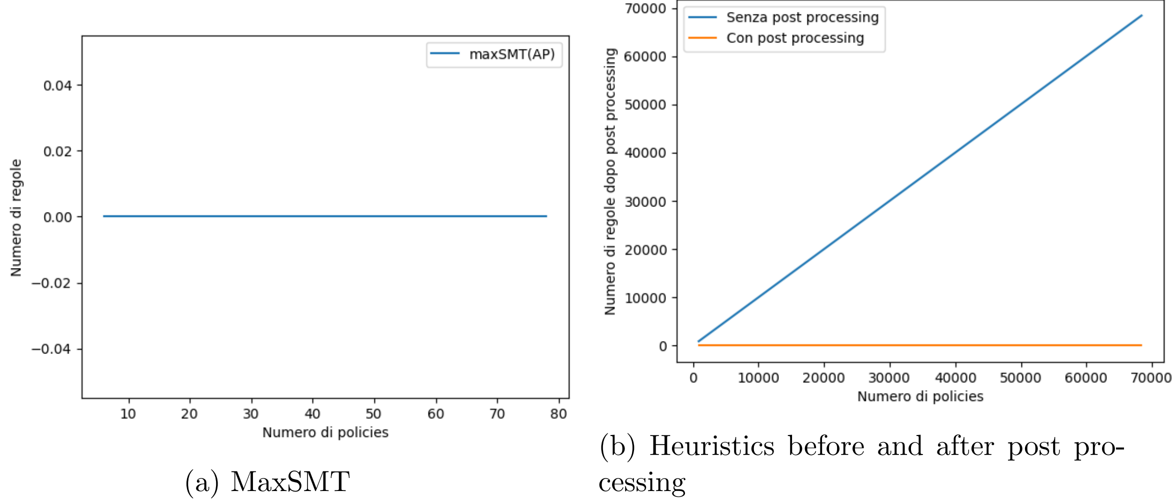
Figure 9.4: Number of allocated firewalls - INTERNET2

Based on images 9.3 and 9.4, it can be observed that, similar to the tests conducted earlier, there are no notable discrepancies between the heuristics and the solver in terms of the number of allocated firewalls. Additionally, there are no significant distinctions between the two types of topology.

9.1.3 Number of Configured Rules

Geant and Internet2 Topologies

The results from the tests conducted on both topologies are combined into a single graph due to their identical nature in this case.



(a) MaxSMT

(b) Heuristics before and after post processing

Figure 9.5: Number of configured rules - GEANT and INTERNET2

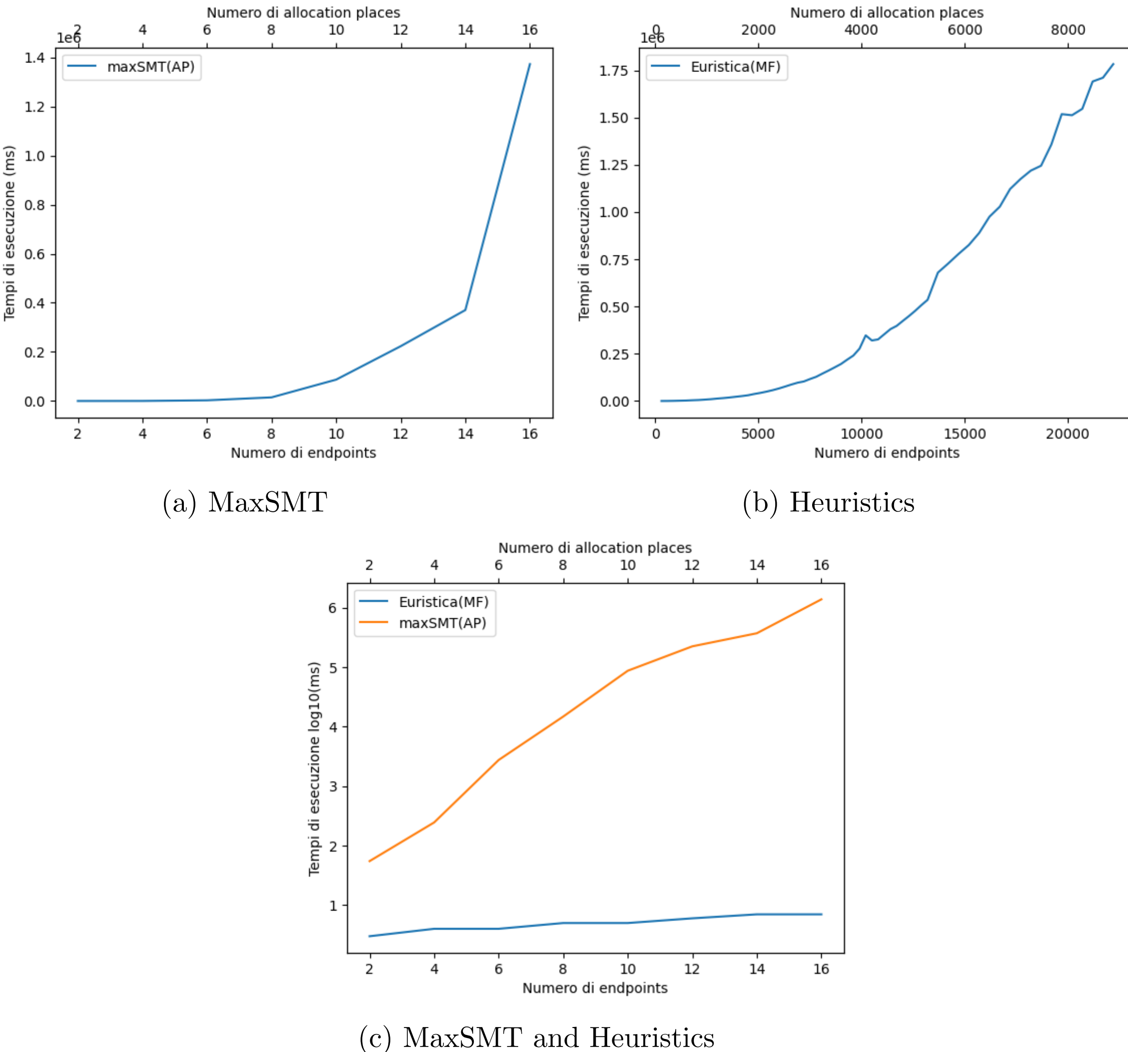
The graph shown in Figure 9.5b clearly shows that the recent implementation of post-processing rules algorithm has aligned the heuristics, in terms of the number of configured rules, with the version based on maxSMT. Specifically, before the implementation of this new algorithm, the heuristics allocated a number of rules equal to the number of requirements when dealing with only isolation requirements. However, after applying this new feature, the firewalls are configured with a single default *DENY* rule, effectively allocating 0 rules (similar to the solver), thus saving a significant amount of memory.

9.2 50% Isolation Requirements and 50% Reachability Requirements

9.2.1 Execution Times

Geant Topology

In general, we notice that the solver's data exhibits a faster growth rate compared to that of the heuristics, reaching the time limits (around half an hour) with a smaller network size than in the previous case where only isolation requirements were present. This smaller network consisted of 16 endpoints and around 50 requirements. Furthermore, it can be observed that the growth rate remains relatively



(a) MaxSMT

(b) Heuristics

(c) MaxSMT and Heuristics

Figure 9.6: Execution times comparison - GEANT

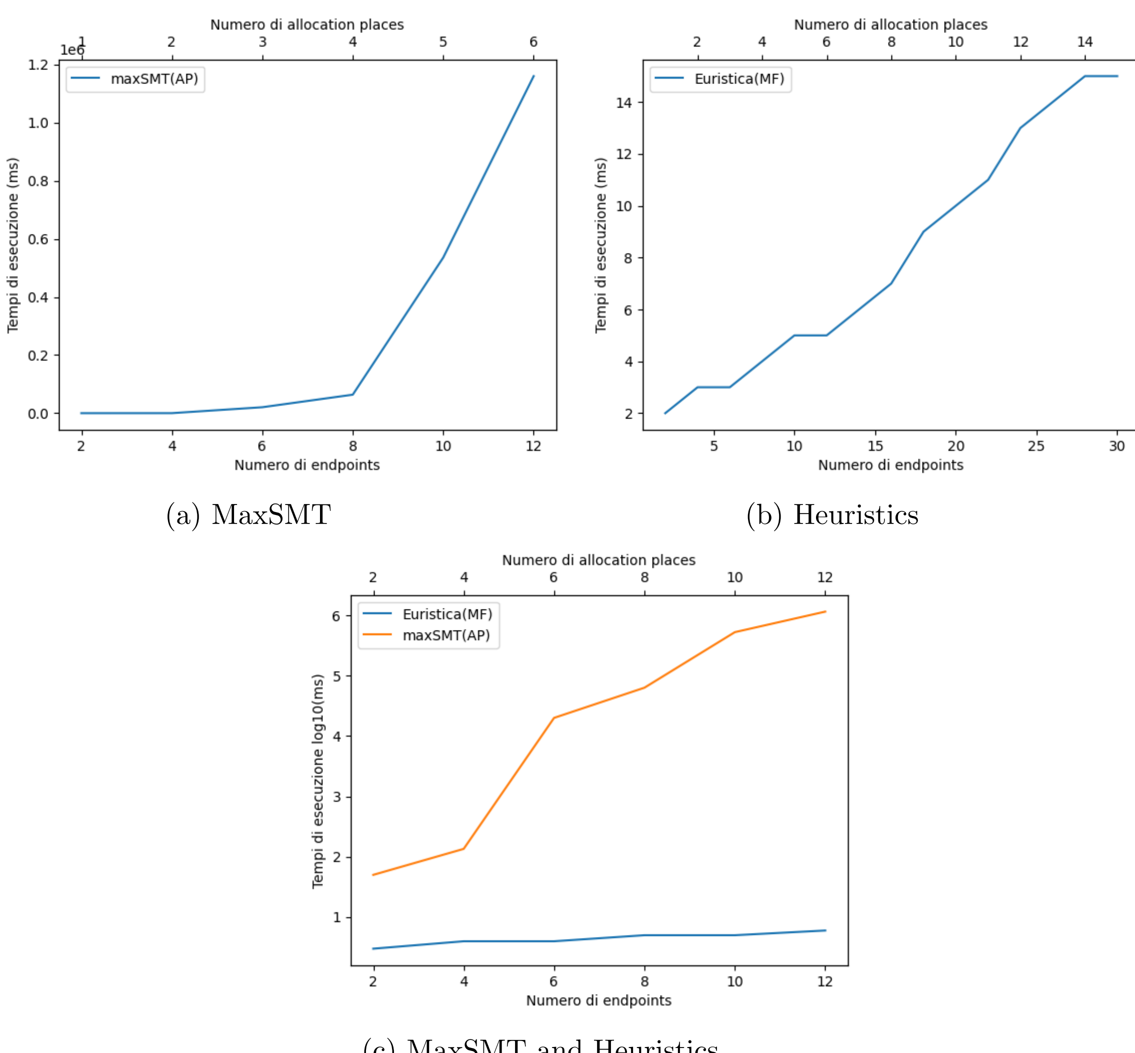
linear up to 10 endpoints. Beyond 10 endpoints, it transitions into an exponential pattern. This outcome highlights the challenges faced by the MaxSMT-based approach when dealing with mixed requirements.

Upon observing the heuristics, it's evident that introducing mixed requirements doesn't affect its performance significantly. The performance remains largely consistent compared to the previous scenario that only included isolation requirements.

In this context, the differences in orders of magnitude are outlined as follows:

- Up to 4 endpoints: a difference of 2 orders of magnitude;
- From 5 to 7 endpoints: a difference of 3 orders of magnitude;
- From 8 to 11 endpoints: a difference of 4 orders of magnitude;
- 12 endpoints and beyond: a difference of 5 orders of magnitude.

Internet2 Topology



(a) MaxSMT

(b) Heuristics

(c) MaxSMT and Heuristics

Figure 9.7: Execution times comparison - INTERNET2

For this topology with mixed requirements, the threshold is reached with a smaller setup of about 12 endpoints and 36 requirements. The increase in time is consistently linear until 8 endpoints but experiences a sharp rise thereafter from 9 endpoints onward.

For the heuristics instead, we can make the same observations as in the previous case. It reaches the time limits with a larger network, consisting of about 25,000 endpoints and 75,000 requirements, thus not being affected by the added complexity of mixed requirements.

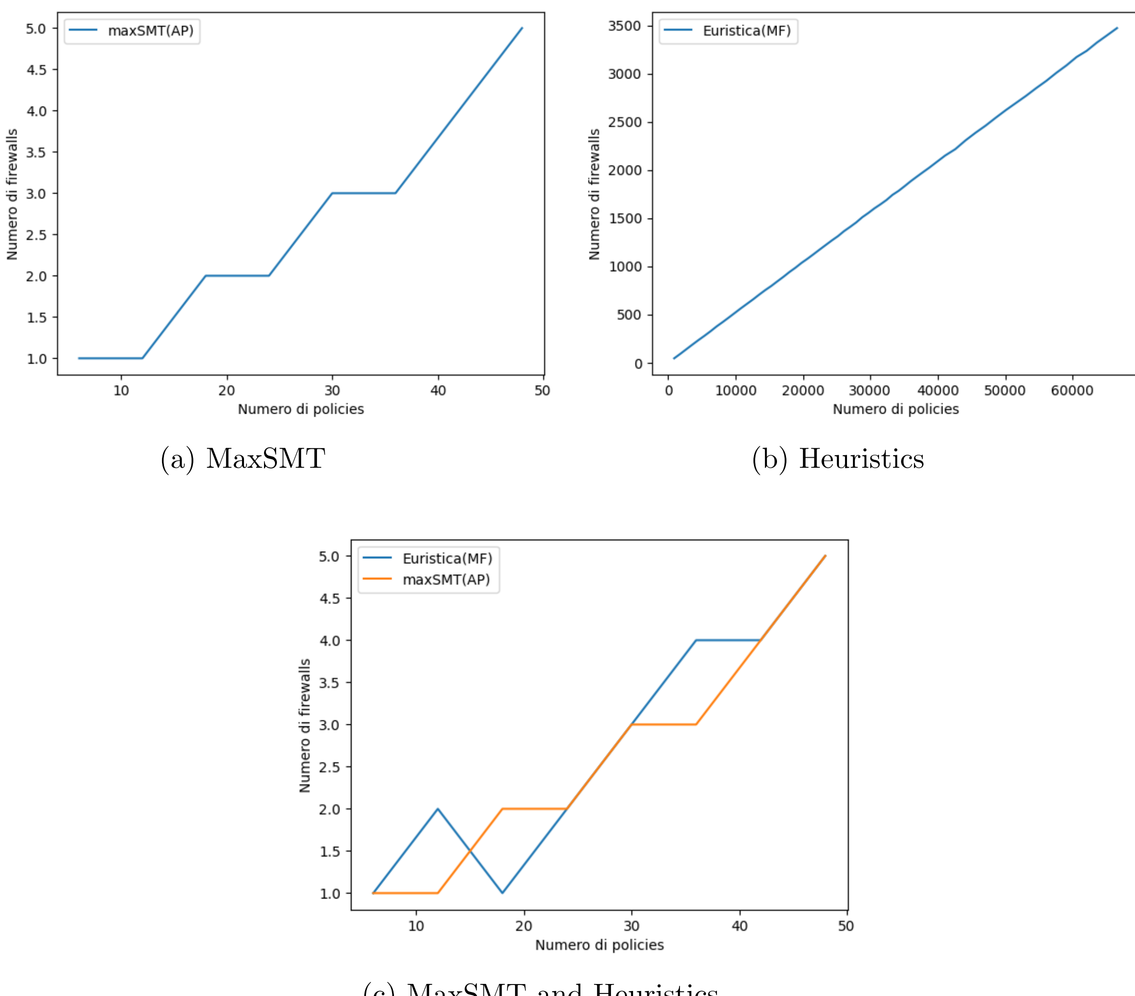
The difference in orders of magnitude is as follows:

- Up to 5 endpoints: 2 orders of magnitude;
- From 6 to 9 endpoints: 4 orders of magnitude;
- 10 to 11 endpoints: 5 orders of magnitude;

- 12 endpoints and above: 6 orders of magnitude.

9.2.2 Number of Allocated Firewalls

Geant Topology



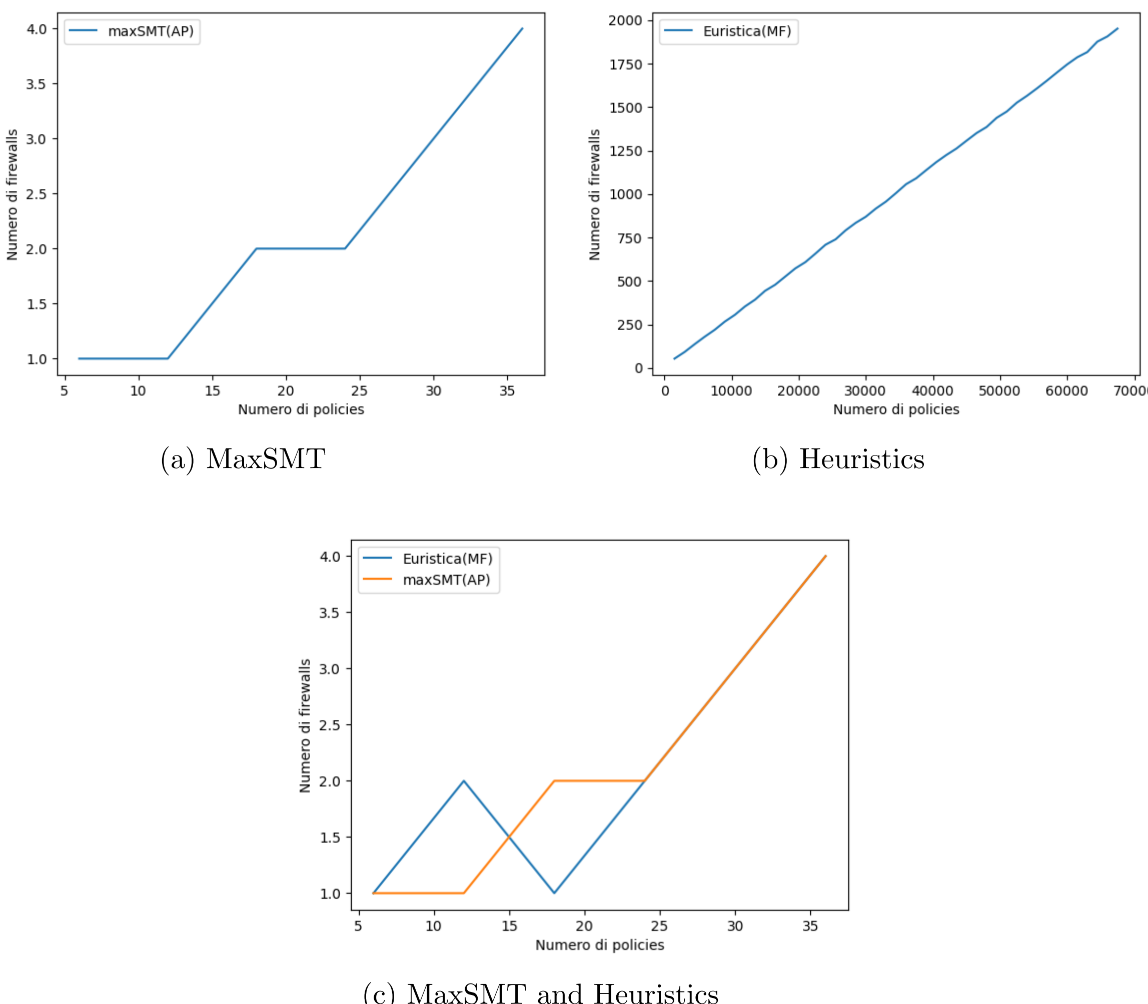
(a) MaxSMT

(b) Heuristics

(c) MaxSMT and Heuristics

Figure 9.8: Number of allocated firewalls - GEANT

Internet2 Topology



(a) MaxSMT

(b) Heuristics

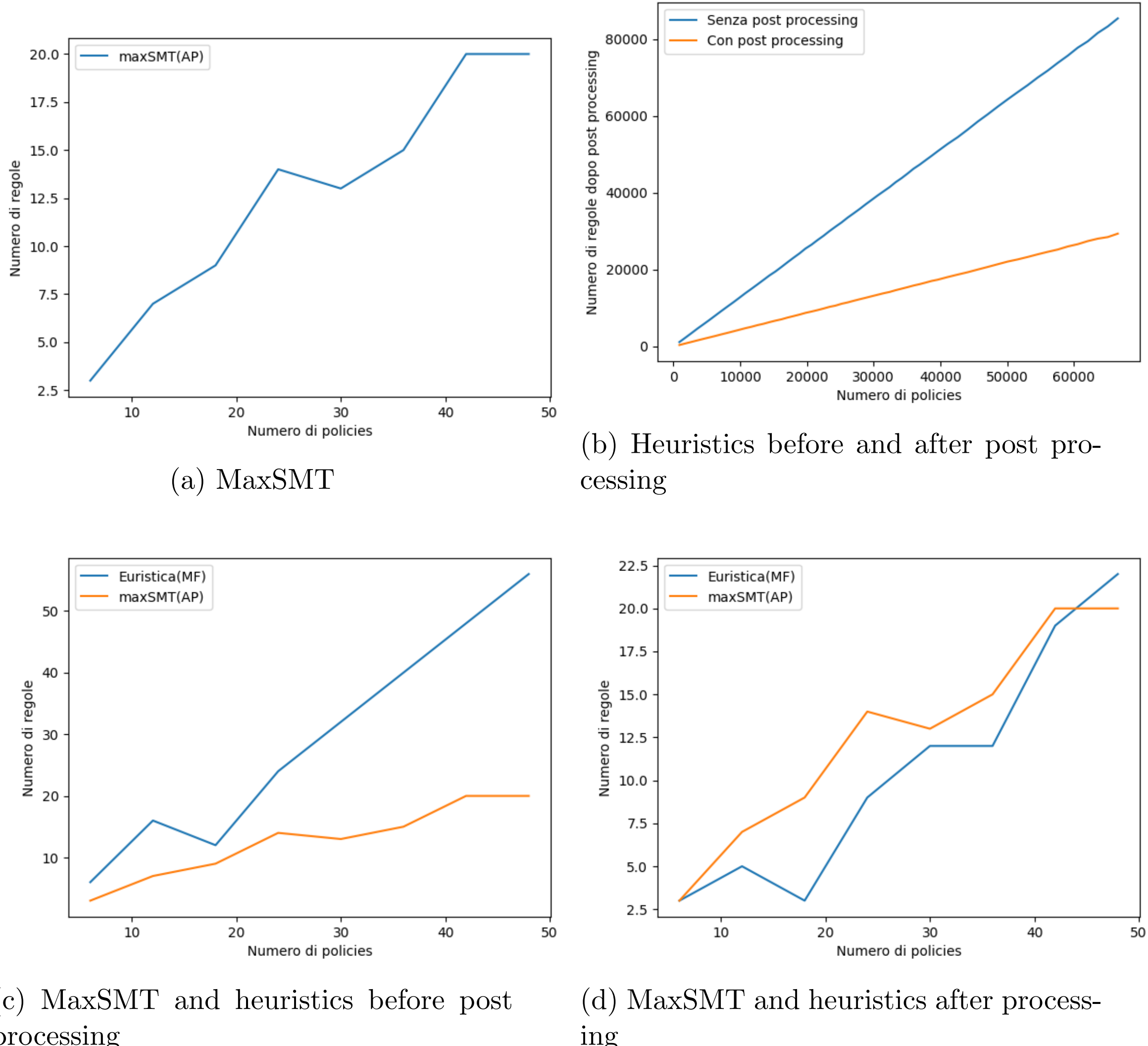
(c) MaxSMT and Heuristics

Figure 9.9: Number of allocated firewalls - INTERNET2

Even in this case when there are mixed requirements, there are not significant differencies in terms on number of allocated firewalls.

9.2.3 Number of Configured Rules

Geant Topology



(a) MaxSMT

(b) Heuristics before and after post processing

(c) MaxSMT and heuristics before post processing

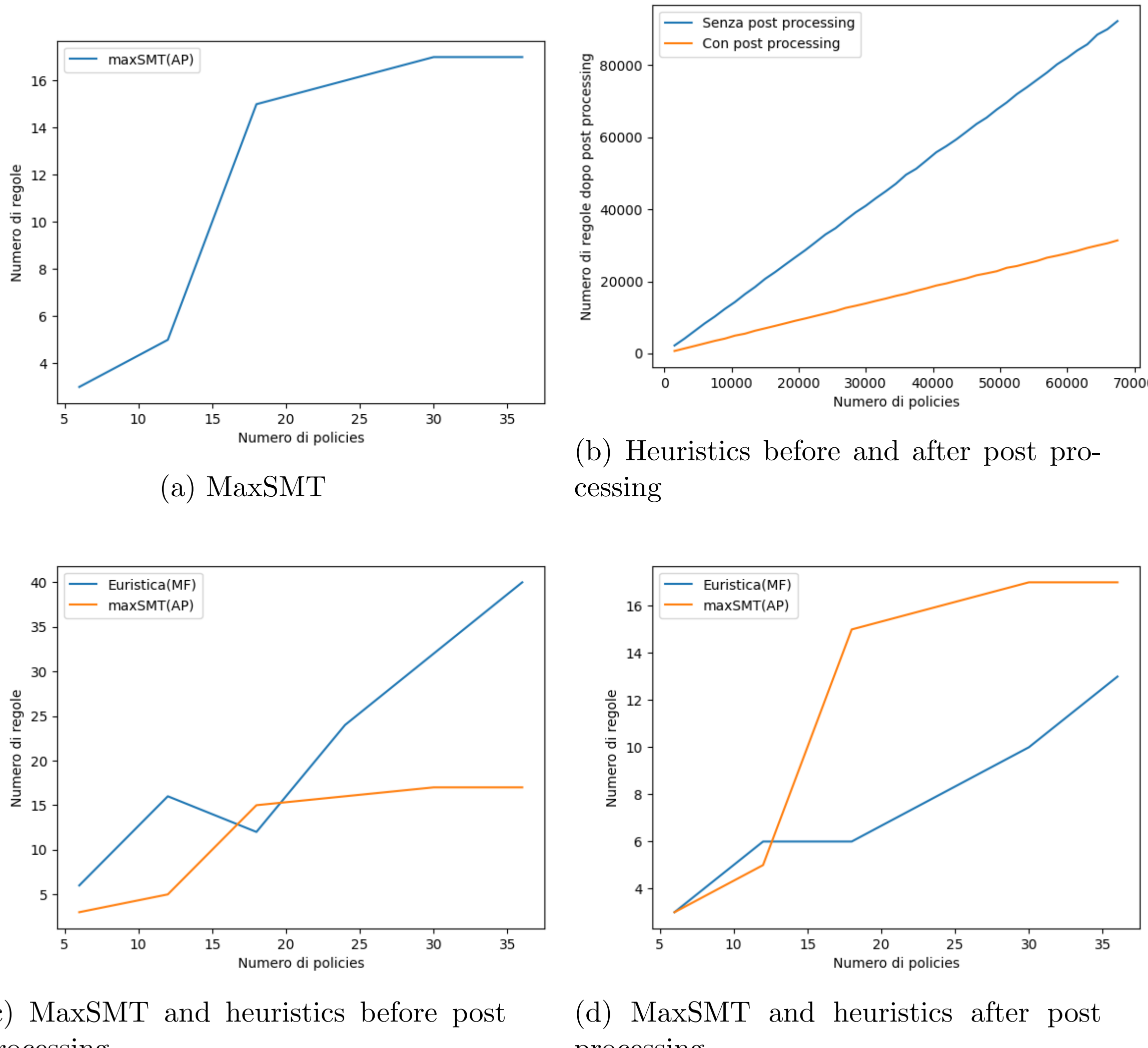
(d) MaxSMT and heuristics after processing

Figure 9.10: Number of configured rules - GEANT

In scenarios involving mixed requirements, including both isolation and reachability, the post-processing function has significantly reduced the number of assigned rules by approximately two-thirds compared to the original configuration. As illustrated in Figure 9.10c, prior to implementing post-processing, the heuristics generated a higher number of rules compared to the solver: from an increase of 100% in networks with few nodes to a 300% increase in networks with around twenty nodes, showing a rising trend. However, with the implementation of this function, it is evident how the heuristics is able to offset this difference by allocating approximately the same number of firewalls. Upon examining Figure 9.10d, an unexpected observation is made: after the post-processing function is applied, the number of rules allocated by the heuristics is actually lower than the number allocated by the solver. This seems peculiar because the solver, which tackles the MaxSMT

problem, typically finds the most optimized solution available. The reason behind this lies in the fact that we used the version based on Atomic Predicates for the solver and the version based on Maximal Flows for the heuristics. The differing modeling approaches for traffic flows lead to the solver allocating slightly more rules than the heuristics, given its reliance on Atomic Predicates (as discussed in Chapter 4). However, further tests regarding this discrepancy will be conducted in the subsequent chapter.

Internet2 Topology



(a) MaxSMT

(b) Heuristics before and after post processing

(c) MaxSMT and heuristics before post processing

(d) MaxSMT and heuristics after post processing

Figure 9.11: Number of configured rules - INTERNET2

The observations made for the *Geant* topology hold true in this scenario as well, where the new post-processing function for the heuristics has reduced the number of allocated rules by approximately 2/3 compared to the original configuration. Additionally, prior to implementing the post-processing function, the heuristics generated a higher number of rules compared to the solver: ranging from a 100%

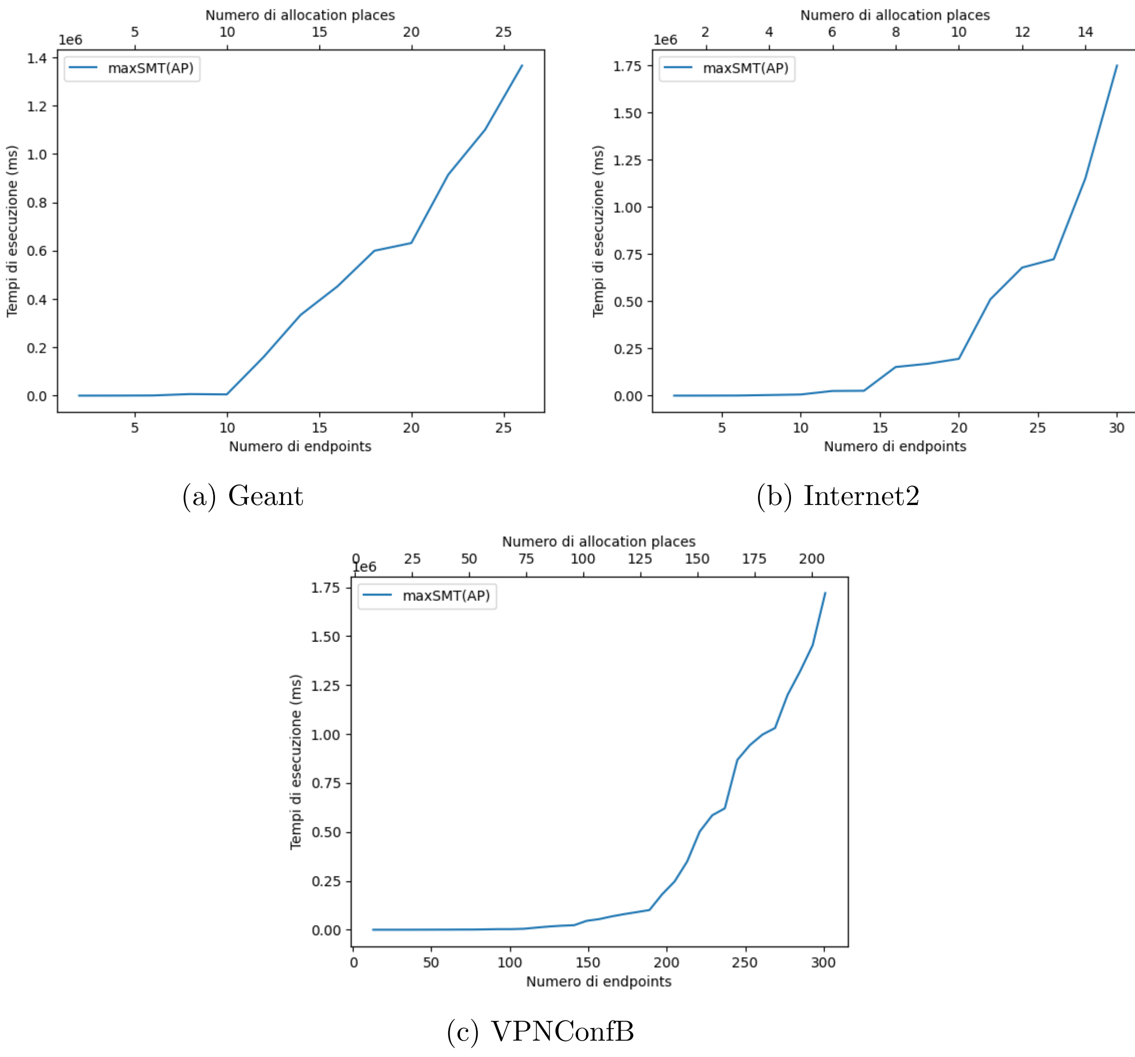
increase to, in this case, a 200% increase, with a growing rate. However, the application of this function clearly shows how the heuristics manages to compensate for this difference by configuring a slightly lower number of rules than the solver, while allocating approximately the same number of firewalls. The reason is the same as before.

9.3 Evaluation of Differences Across Topologies

This section shifts its focus towards analyzing the performance and optimality differences among the three examined topologies. The objective is to assess whether there are variations in the framework's execution based on the examined topology.

9.3.1 MaxSMT Version - 100% Isolation Requirements

Execution Times

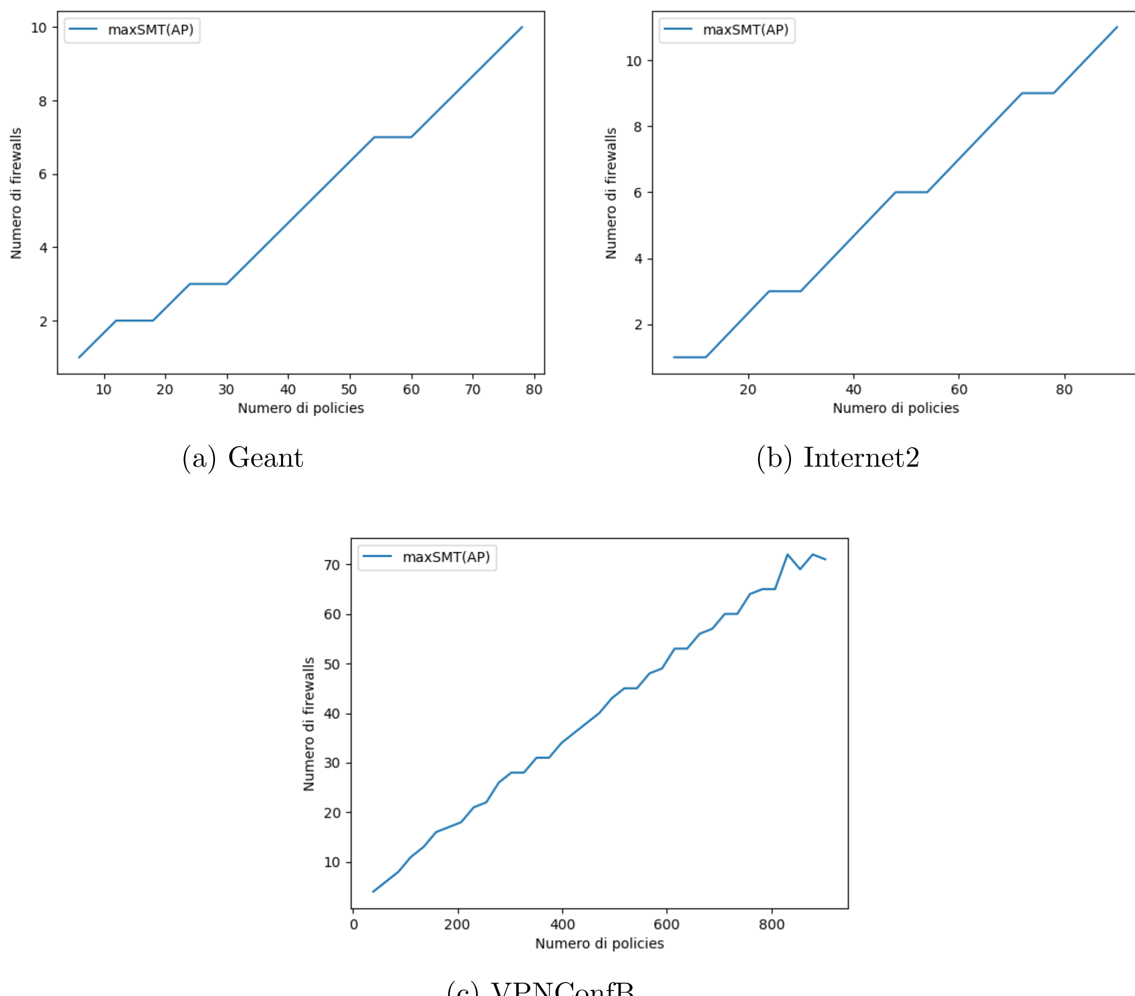


(a) Geant (b) Internet2

(c) VPNConfB

Figure 9.12: Execution Times

Number of Allocated Firewalls



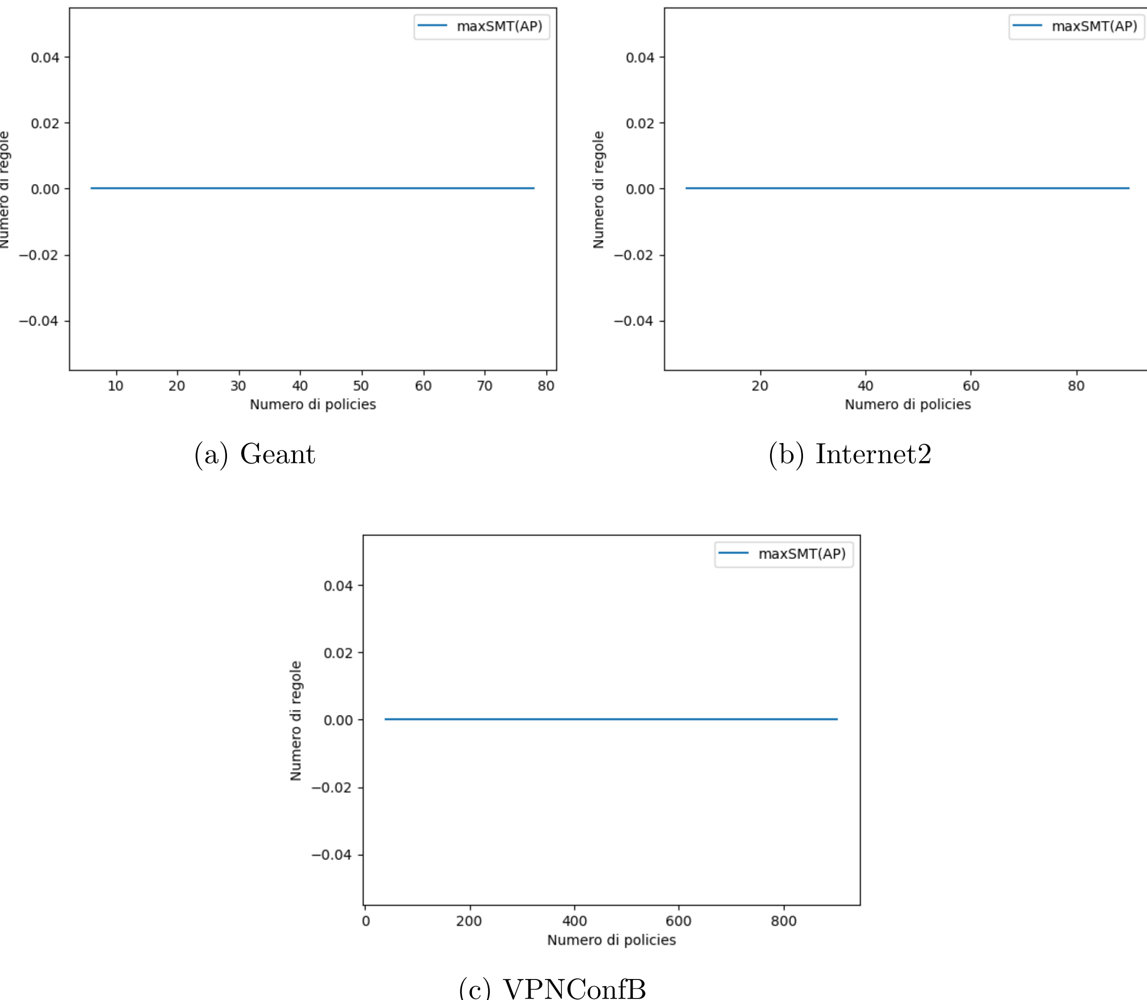
(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.13: Number of Allocated Firewalls

Number of Configured Rules



(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.14: Number of Configured Rules

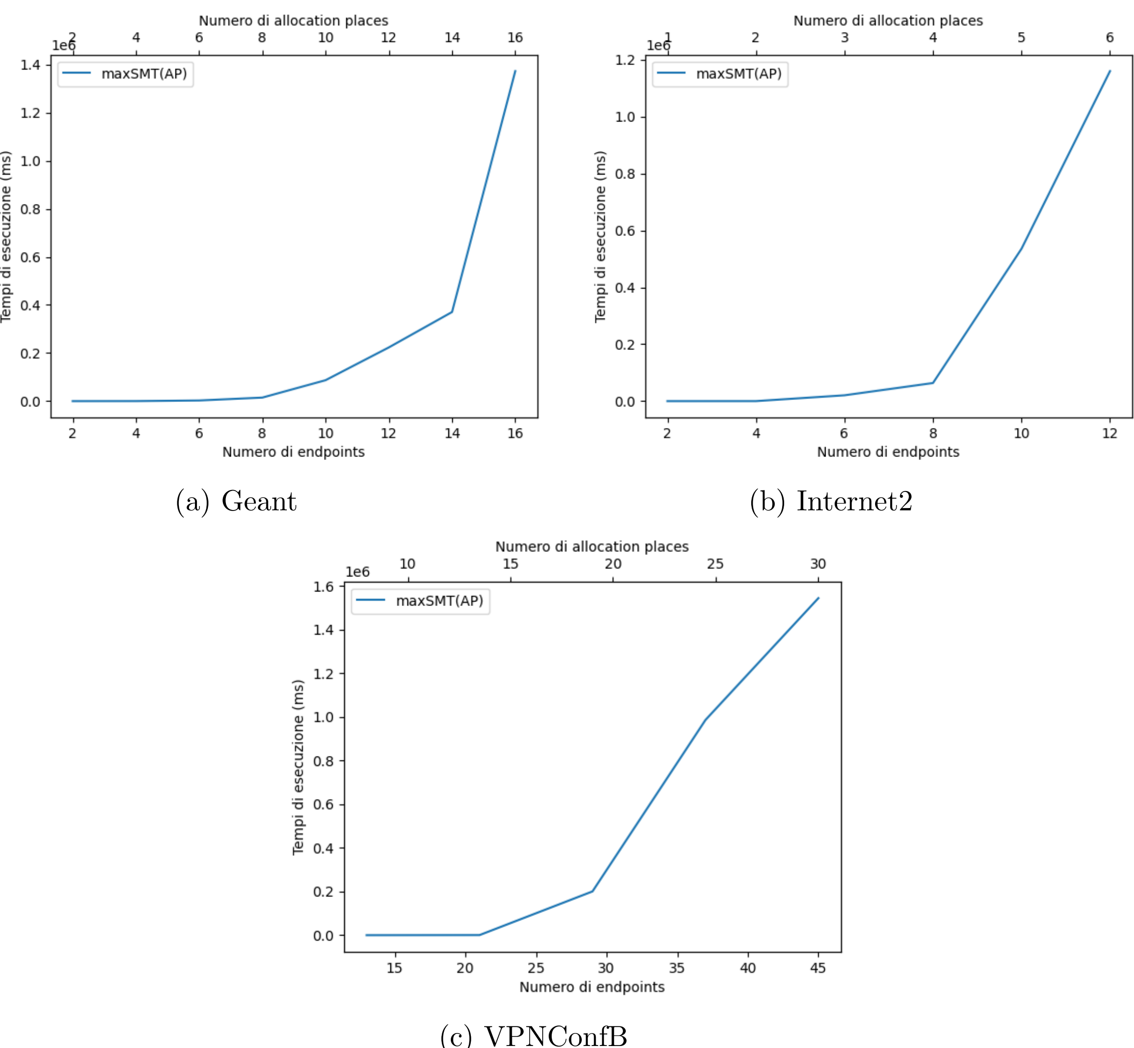
The graphs provided indicate a similarity in performance between the *Geant* and *Internet2* topologies. There is a slight trend where *Internet2* seems to exhibit slightly better performance than *Geant*, although the difference is not substantial.

On the other hand, there is a significant difference observed with *VPNConfB*, as this configuration reaches time limits with much larger networks, consisting of approximately 300 endpoints and 900 policies (about ten times larger than the other 2 topologies). It is important to note that the tests for *VPNConfB* did not account for ports in the policies. Consequently, the more complex the policies become, the greater the challenge for the solver to process them.

When it comes to the number of allocated firewalls and rules, there are no significant differences observed among the three different topologies.

9.3.2 MaxSMT Version - 50% Isolation Requirements and 50% Reachability Requirements

Execution Times



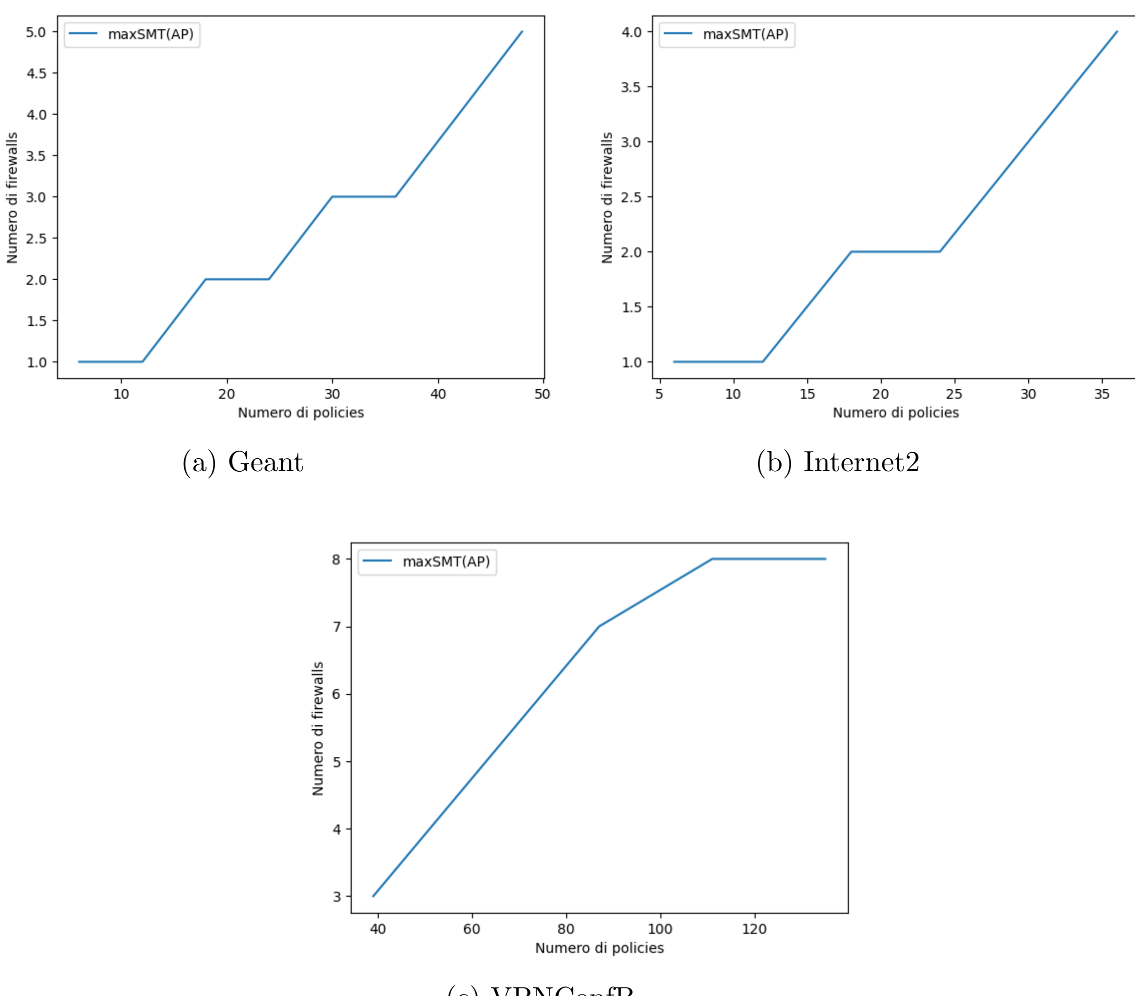
(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.15: Execution Times

Number of Allocated Firewalls



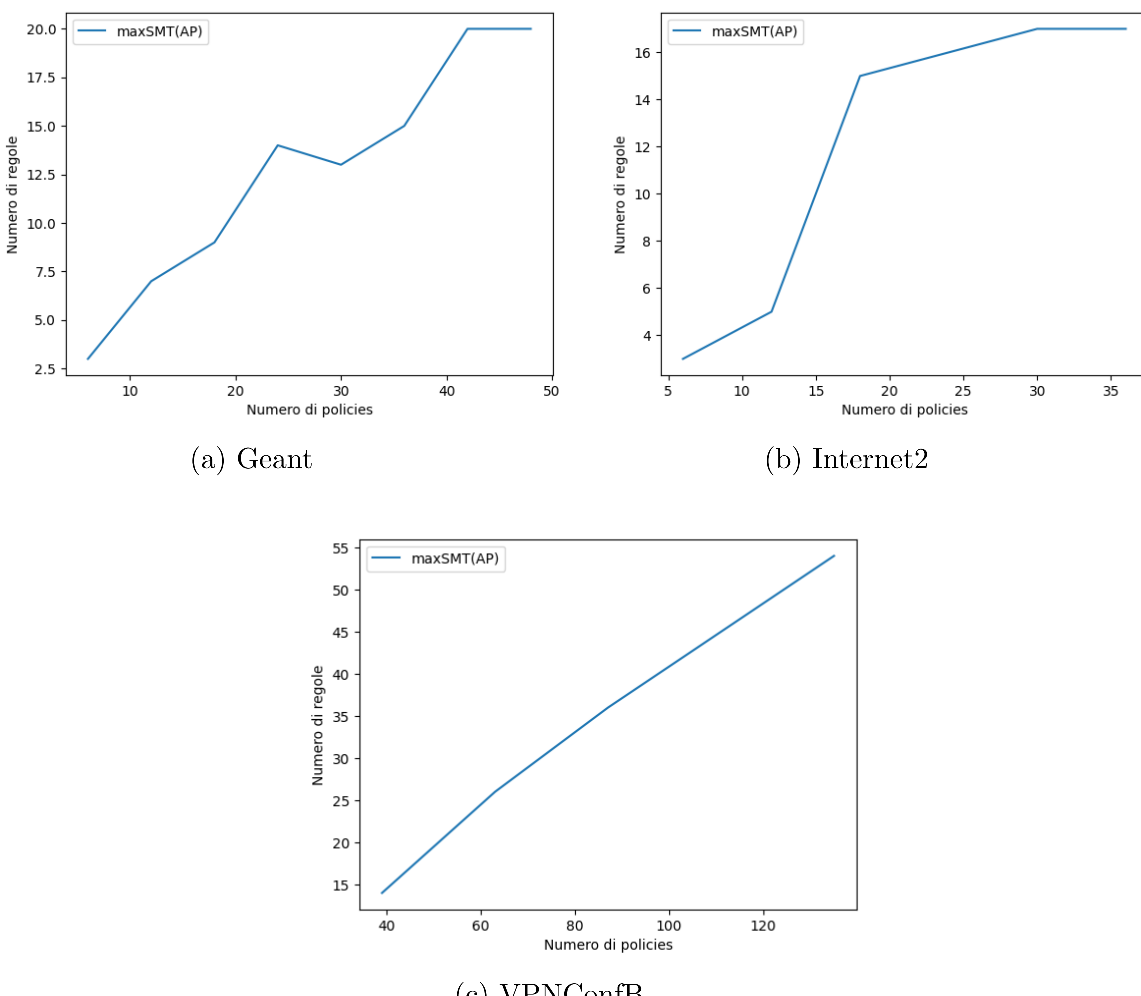
(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.16: Number of Allocated Firewalls

Number of Configured Rules



(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.17: Number of Configured Rules

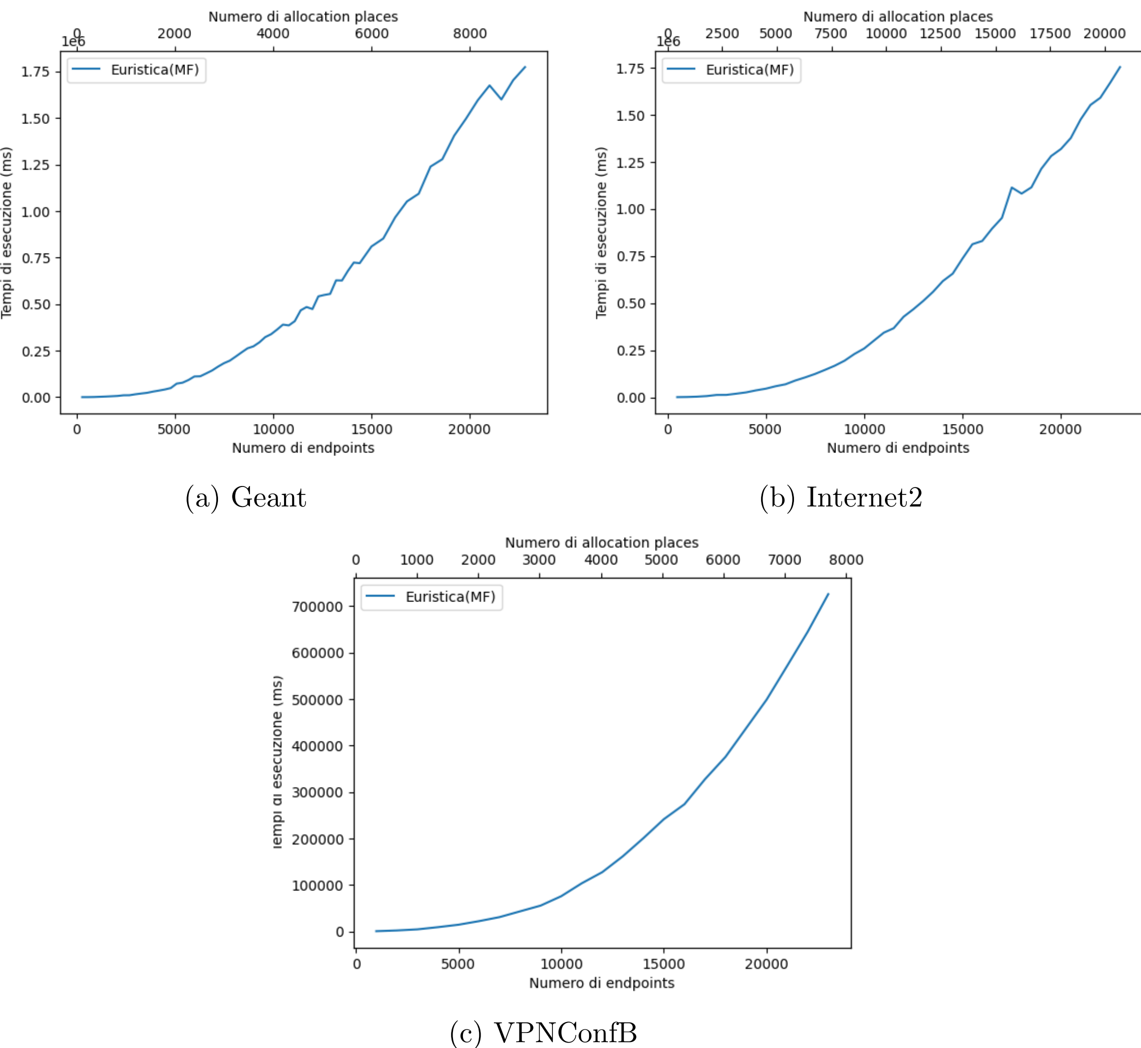
Considering mixed requirements, we observe a convergence in differences among the various topologies. Both the *Geant* and *Internet2* topologies reach their limits with slightly smaller networks compared to the previous case, comprising approximately 16 endpoints and 48 policies for *Geant* and 12 endpoints and 36 policies for *Internet2*. However, Similarly, *VPNConfB* also reaches its limits but much earlier than in the previous case, transitioning from 300 endpoints and 900 policies to around 45 endpoints and 135 policies.

These findings underscore how the incorporation of reachability requirements, alongside isolation requirements, markedly complicates the solver's task in rule configuration.

Similarly, in this case, there are no significant differences among the three different topologies regarding the number of firewalls and rules allocated.

9.3.3 Heuristics Version - 100% Isolation Requirements

Execution Times



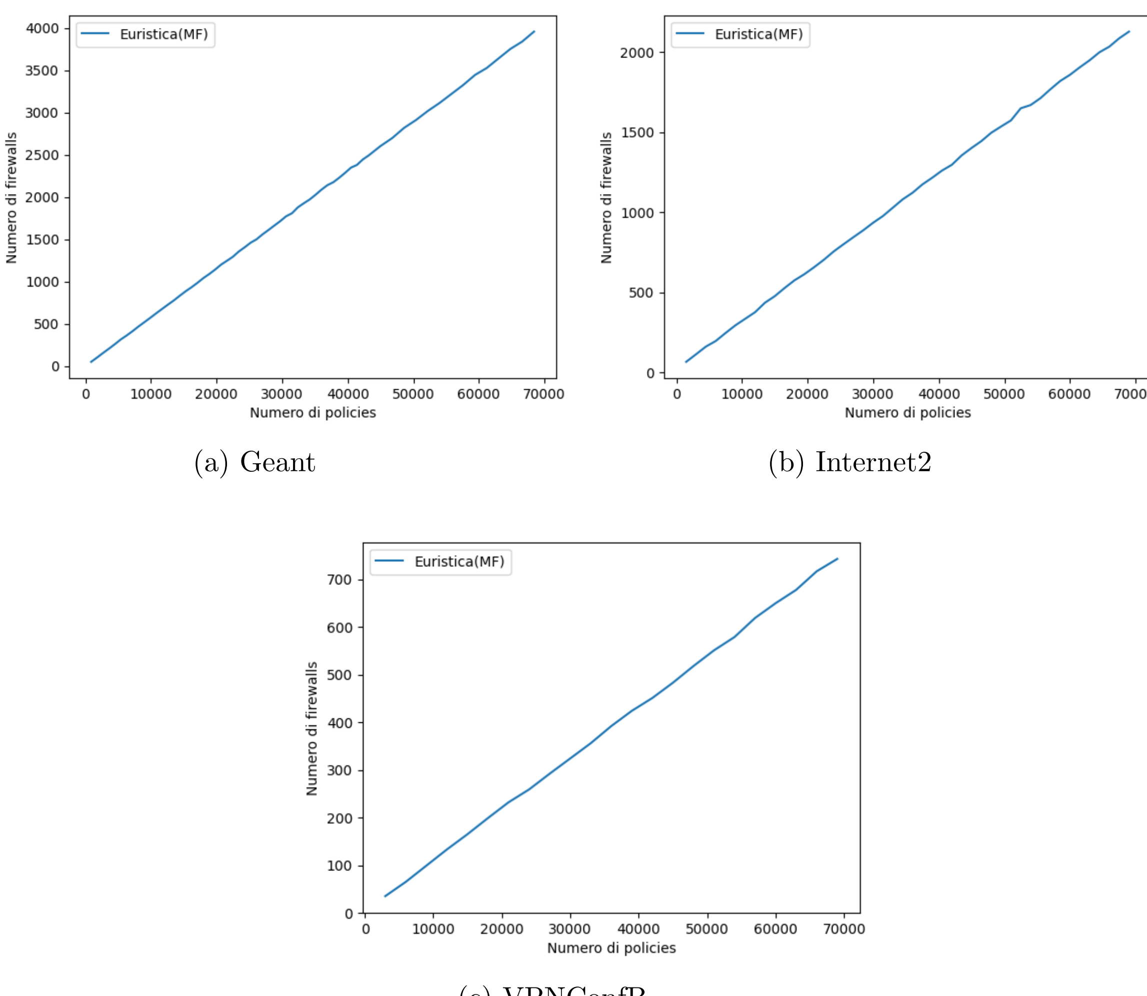
(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.18: Execution Times

Number of Allocated Firewalls



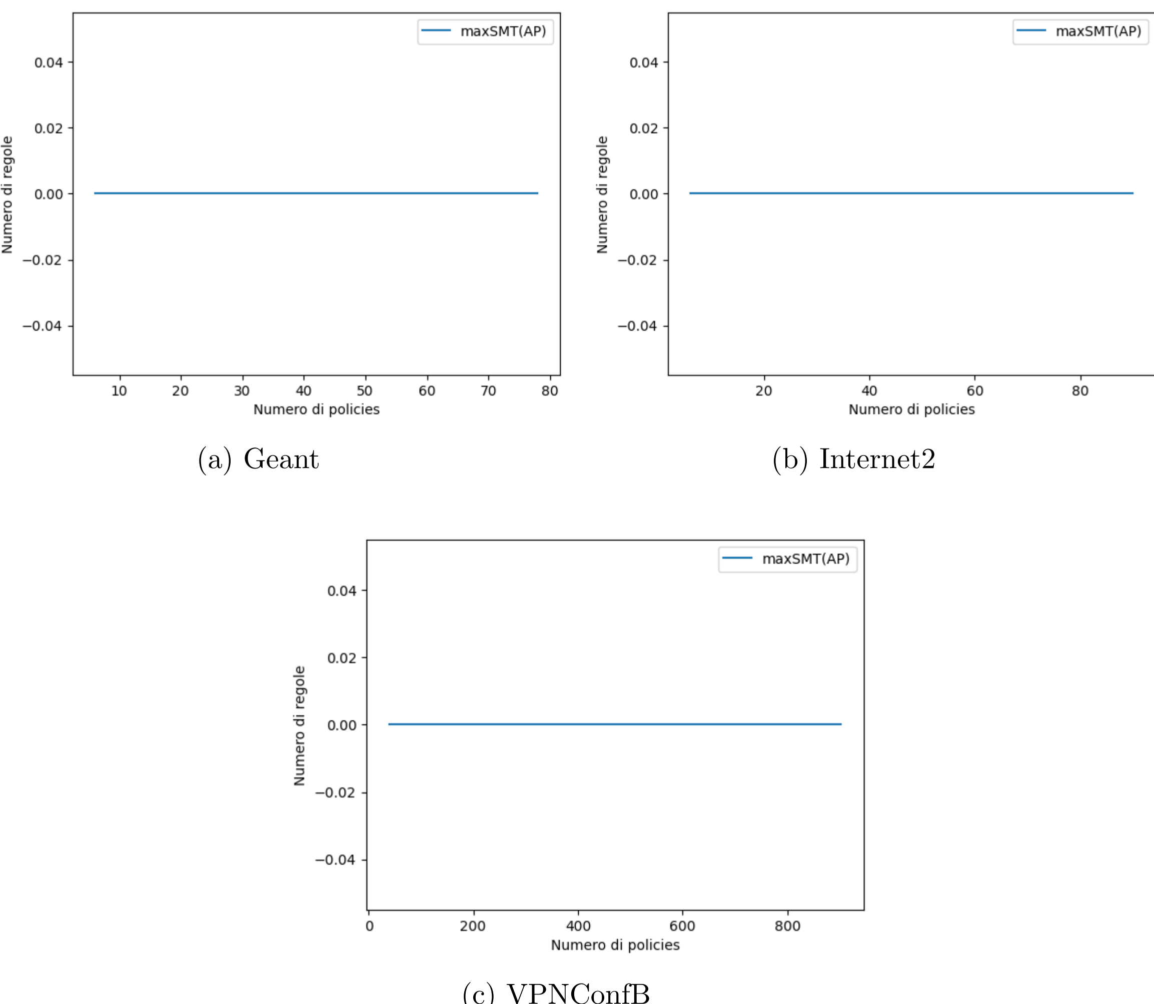
(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.19: Execution Times

Number of Configured Rules



(a) Geant

(b) Internet2

(c) VPNConfB

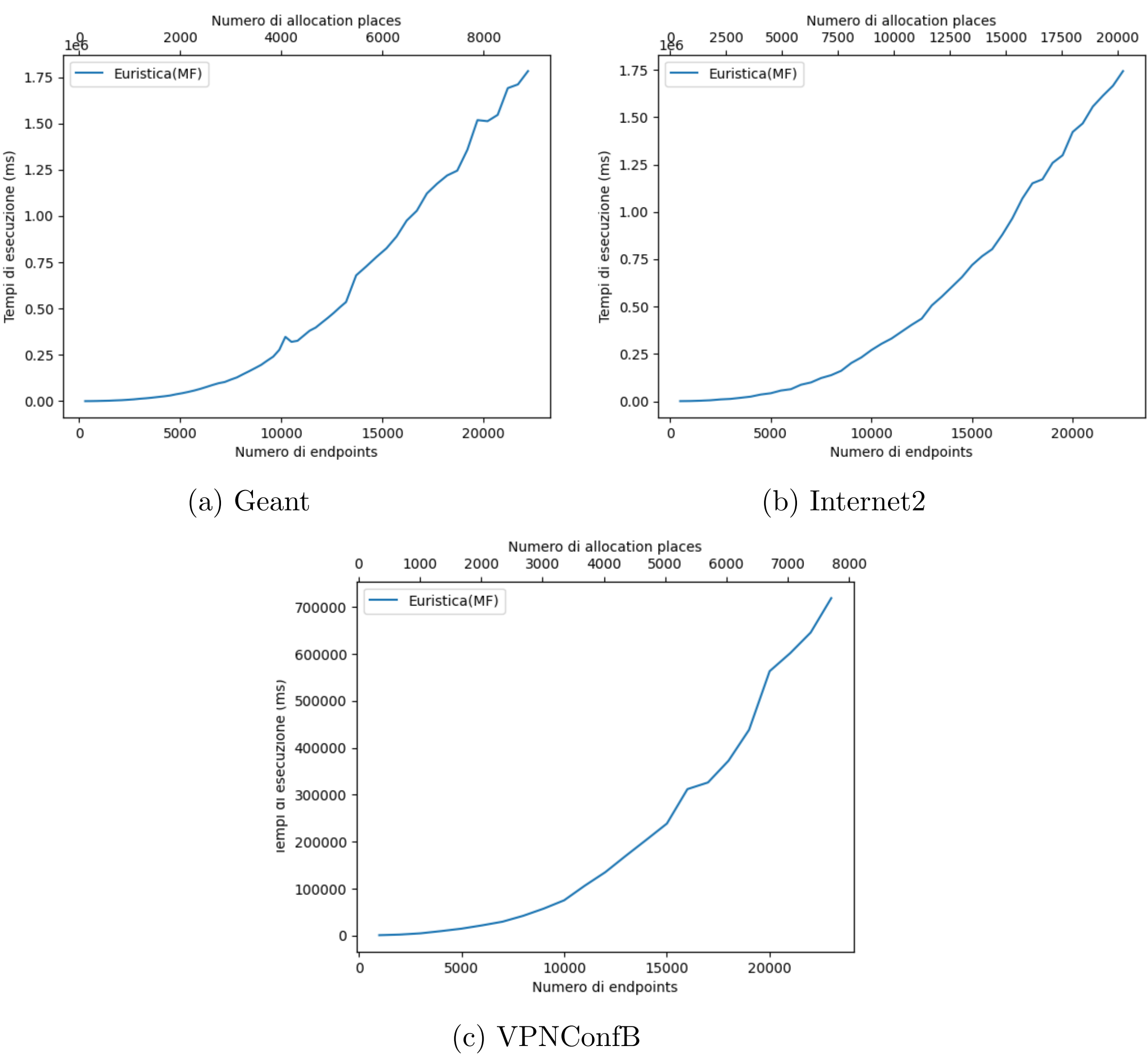
Figure 9.20: Number of Configured Rules

Regarding the heuristics, the *Geant* and *Internet2* topologies show essentially the same execution times. In contrast, *VPNConfB* stands out for faster execution times but reaches Java memory limits when dealing with networks of around 20,000 endpoints.

In terms of number of configured firewalls, there is a difference among the three topologies: *Geant* allocates the most, followed by *Internet2* which allocates almost half of them and *VPNConfB*, which instead allocates about 1/5 of the firewalls allocated in the *Geant* topology. For rules, however, all three topologies allocate the same number under the same number of requirements.

9.3.4 Heuristics Version - 50% Isolation Requirements and 50% Reachability Requirements

Execution Times



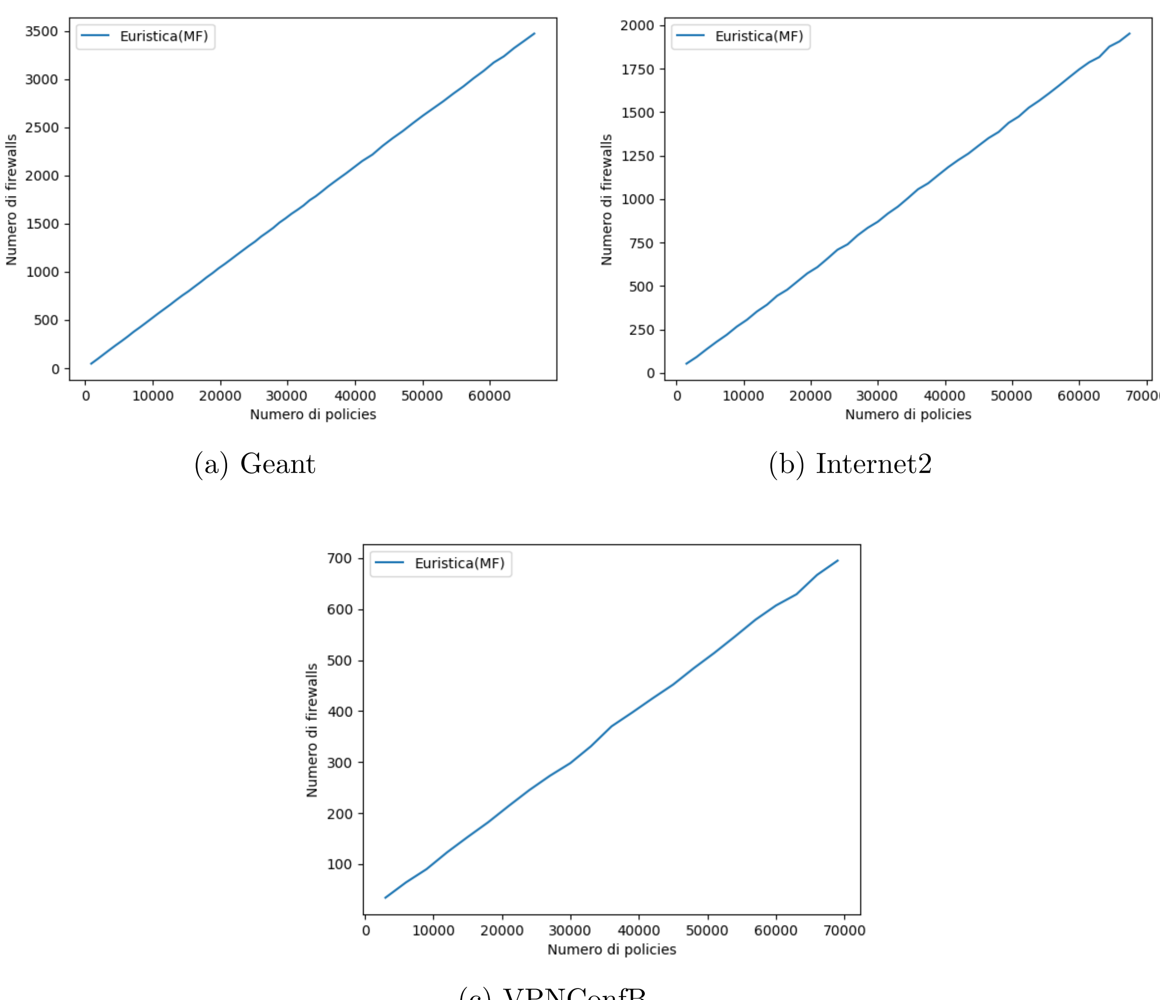
(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.21: Execution Times

Number of Allocated Firewalls

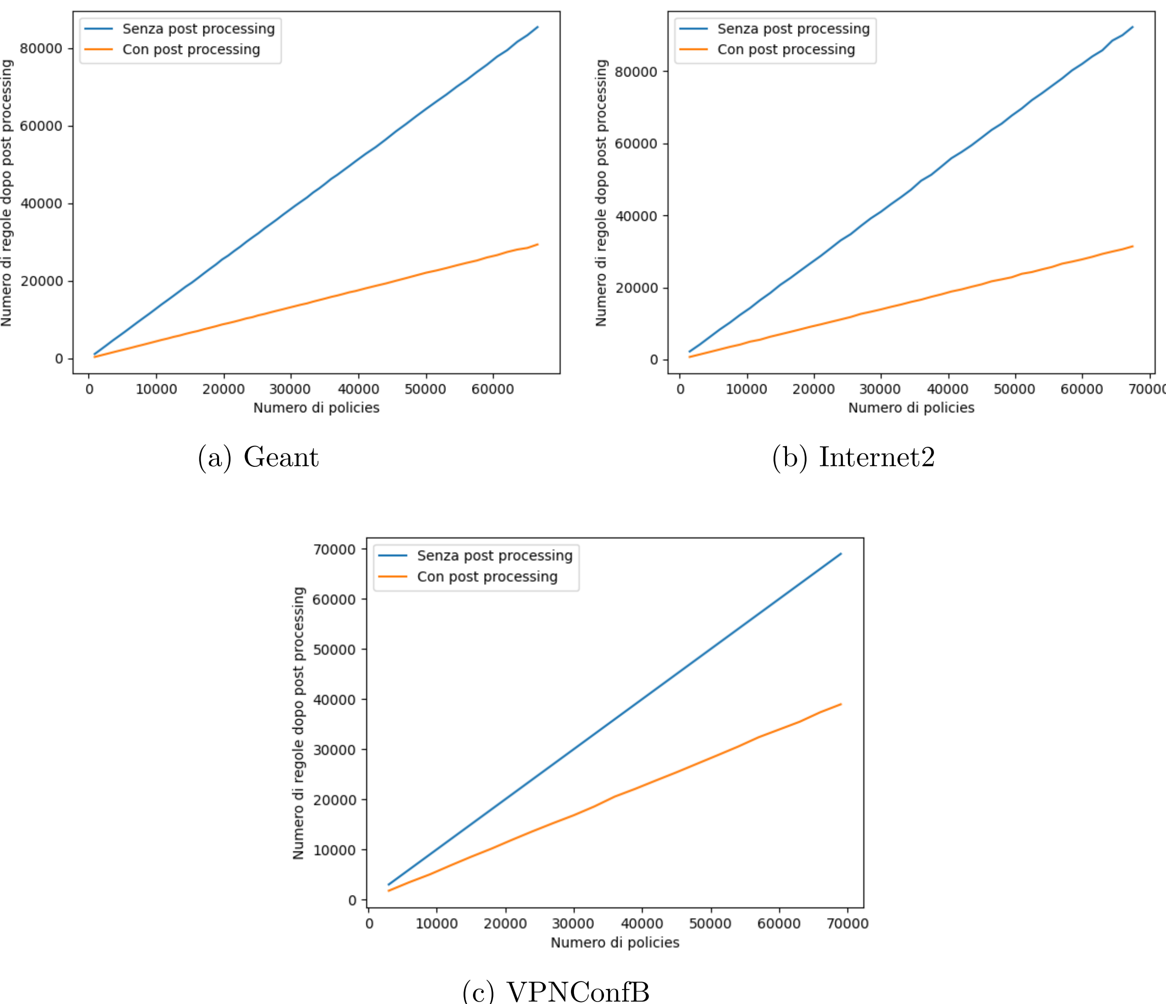


(a) Geant (b) Internet2

(c) VPNConfB

Figure 9.22: Execution Times

Number of Configured Rules



(a) Geant

(b) Internet2

(c) VPNConfB

Figure 9.23: Number of Configured Rules

In the case of the heuristics, both the *Geant* and *Internet2* topologies demonstrate similar execution times. On the other hand, *VPNConfB* topology exhibits faster execution times but encounters memory limitations associated with Java when handling networks comprising approximately 20,000 endpoints.

Concerning the number of configured firewalls, there is a distinction among the three topologies: *Geant* has the highest number, followed by *Internet2* and *VPNConfB* like in the previous case. However, all three topologies allocate an equal number of rules under similar requirements.

In the case of mixed requirements, there are no notable differences compared to scenarios with solely isolation requirements. This outcome underscores that unlike the solver, the heuristics doesn't experience the heightened complexity from reachability requirements, thus maintaining a consistent performance level even when only isolation requirements are present.

9.4 Considerations

In this second phase of testing, which was primarily aimed at evaluating the optimization benefits stemming from the post-processing algorithm's implementation for the heuristics and discerning distinctions across different topologies, the following results are achieved:

- **Impact of the post-processing algorithm**: The algorithm has notably reduced the number of rules allocated by the heuristics. Specifically, when the entire set of NSRs comprises isolation requirements, the algorithm ensures that all firewall rules are replaced by a single default DENY rule. This aligns the heuristic's behavior more closely with that of the solver. However, in cases involving mixed requirements (both isolation and reachability), one of the rule sets is replaced by a default rule, resulting in about a two-thirds reduction in the number of configured rules.

- **Performance disparities**: As noted in the initial testing in Chapter 7, there are significant performance variations between the heuristics and the solver. These differences range from a minimum of 2 orders of magnitude for very small networks with a few endpoints to 5 orders of magnitude for networks with tens of endpoints, with performance disparity increasing as network size grows.

- **Distinctions between topologies**: Notably, when considering the solver and factoring in ports in policies, there's a substantial increase in computational complexity. The solver reaches its half-hour limit with networks approximately one-tenth the size compared to scenarios where ports are excluded from policies. Conversely, the heuristics can handle much larger networks, but introducing ports in policies leads to roughly a threefold increase in execution times. Regarding the number of configured rules, the graph trends exhibit a relatively linear pattern, with no significant deviations observed among the three topologies. Instead, there are differencies when considering the number of allocated firewalls: *Geant* allocates the highest number of firewalls, followed by *Internet2* and *VPNConfB*. This depends by the morfology of the topologies and by the number of Allocation Places that are present in them.

Chapter 10

Test Campaign - Third Phase

This final testing chapter focuses on evaluating the results obtained from performance and optimality tests, with particular emphasis on two key parameters:

1. **Degree of Optimality**: the first section aims to precisely evaluate the difference in the number of allocated firewalls and configured rules among the following versions of the framework:

   - Heuristics based on Maximal Flows
   - MaxSMT based on Maximal Flows

   In Chapter 9, the discrepancy observed when considering the number of configured rules stemmed from testing two different versions: the heuristic version was built on Maximal Flows, whereas the MaxSMT version was based on Atomic Predicates. Due to the Atomic Predicates approach resulting in configuring a higher number of rules, it was evident that the MaxSMT version configured more rules than the heuristics. Furthermore, the tested topologies, although matching in terms of endpoints and requirements numbers, had random requirements. It means that the set of NSRs was not the same for them. Consequently, the evaluation produced a general result rather than a precise one. This section addresses this issue by testing identical topologies with the same set of requirements across the two versions. The aim is to primarily compare the heuristic version based on Maximal Flows with the MaxSMT version, which is now also based on Maximal Flows rather than on Atomic Predicates as in the previous chapter, to ensure the correct allocation of firewalls and configured rules.

2. **Limits of the Heuristics**: furthermore, the limit that the heuristics can reach in terms of network size and maximum number of requirements before exhausting the memory of the IDE used for the tests, in this case Eclipse, has also been tested.

In this context, the goal is to analyze and compare the performance and optimization capabilities of different framework implementations accurately, as well as to determine the practical limits of the heuristics in terms of scalability and memory management.

10.1 Optimality Tests

During this testing phase, two specific topologies were considered:

1. *Geant*
2. *Internet2*

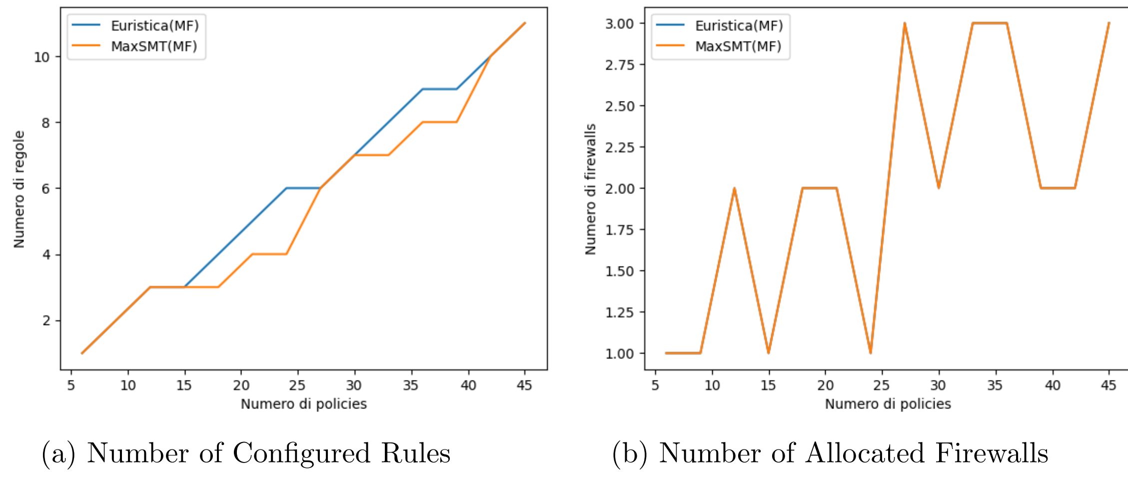
Additionally, three different scenarios were examined to assess the differences in the distribution of requirements:

1. 25% Isolation Requirements and 75% Reachability Requirements
2. 50% Isolation Requirements and 50% Reachability Requirements
3. 75% Rsolation Requirements and 25% Reachability Requirements

For each scenario, both the number of allocated firewalls and the number of rules configured by the framework (after the new implemented post-processing algorithm for the heuristics) were evaluated. Specifically, a comparison was made between the heuristics-based version (Maximal Flows) and the MaxSMT one (Maximal Flows).

10.1.1 Geant Topology

25% Isolation Requirements and 75% Reachability Requirements

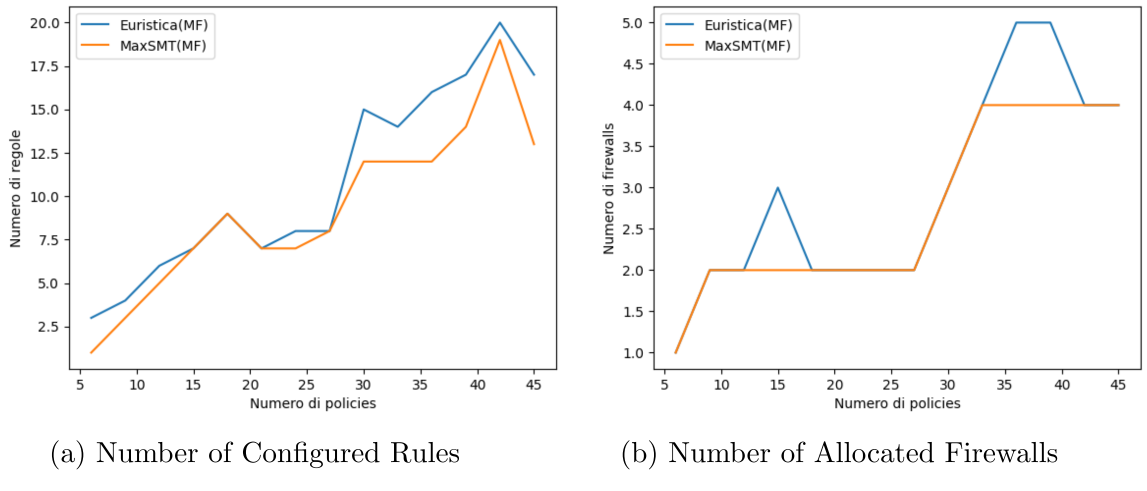


(a) Number of Configured Rules (b) Number of Allocated Firewalls

Figure 10.1

From Figure 10.1a, it can be observed that the heuristic allocates a number of rules greater than or at most equal to the solver. The average difference is approximately 10% more rules allocated, with a maximum difference of 30%. However, in Figure 10.1b, it is evident that the heuristics allocates an equal number of firewalls compared to the MaxSMT.

50% Isolation Requirements and 50% Reachability Requirements

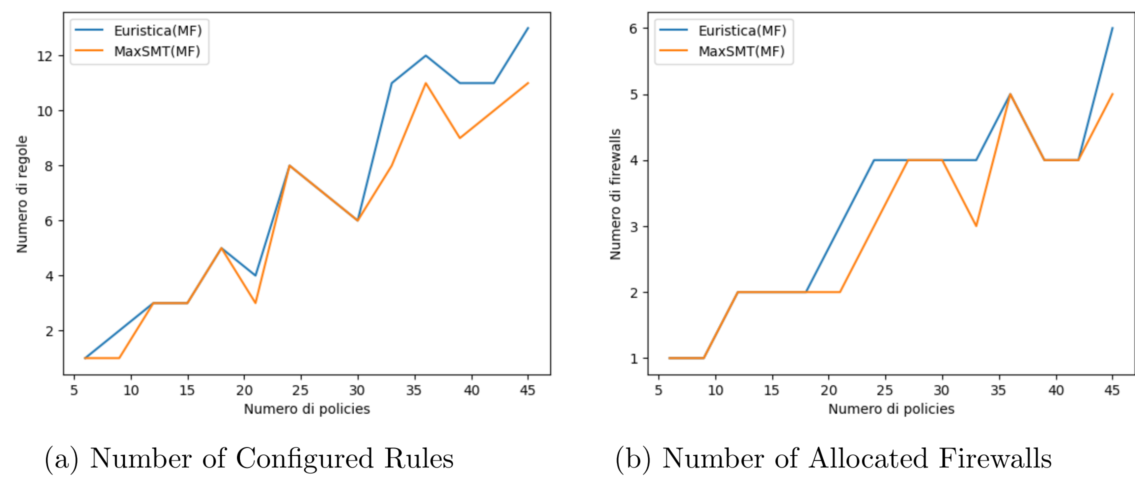


(a) Number of Configured Rules (b) Number of Allocated Firewalls

Figure 10.2

In this case as well, the heuristic approach allocates an equal or higher number of rules compared to the MaxSMT approach, but the difference is more pronounced. In fact the average difference is approximately 16% more rules allocated with a maximum difference of 66% (Figure 10.2a). Even the number of firewalls allocated by the heuristics is equal or slightly higher than those allocated by the MaxSMT this time, with an average difference of 5% more. (Figure 10.2b).

75% Isolation Requirements and 25% Reachability Requirements



(a) Number of Configured Rules (b) Number of Allocated Firewalls

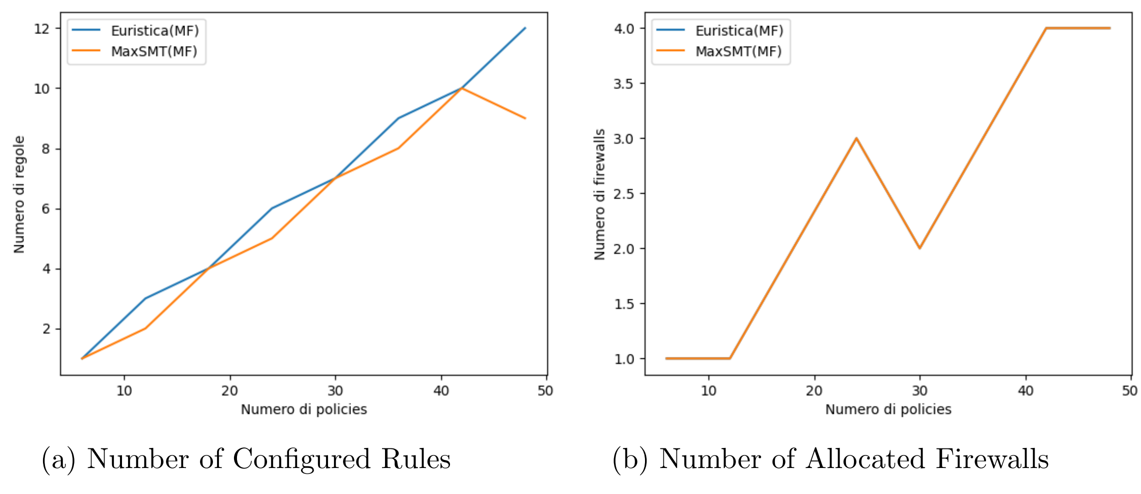
Figure 10.3

In this scenario, the gap between the heuristic approach and the MaxSMT approach narrows. The heuristic method allocates an average of 11% more rules, with a maximum difference of 50% (Figure 10.3a). Similarly, regarding firewalls,

the heuristic allocation remains equal or slightly higher than MaxSMT, with an average difference of 7% more firewalls. (Figure

10.1.2 Internet2 Topology

25% Isolation Requirements and 75% Reachability Requirements

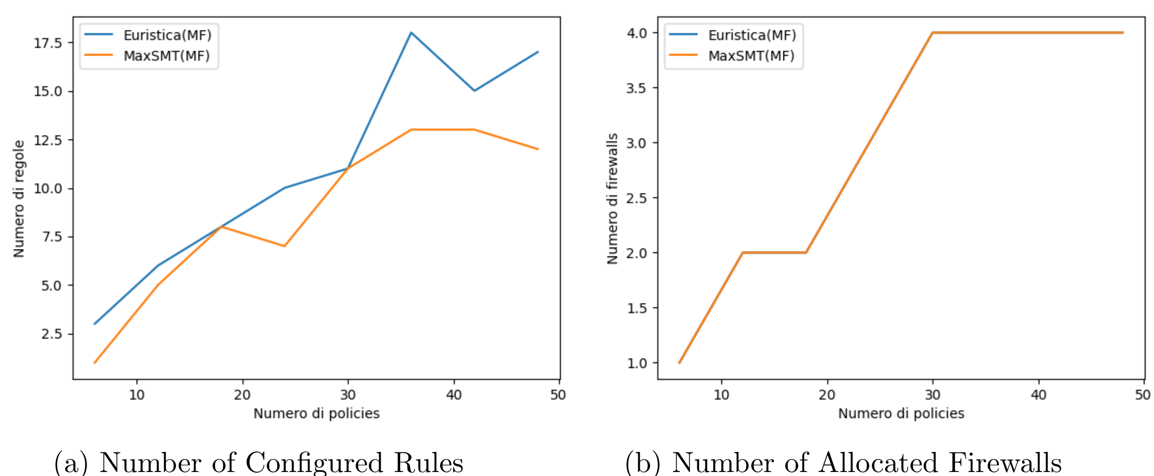


(a) Number of Configured Rules (b) Number of Allocated Firewalls

Figure 10.4

The heuristics allocates an equal or higher number of rules compared to the MaxSMT, with an average difference of 11% more rules and a peak of 33% (Figure 10.4a). For the firewalls, the number allocated by both versions is equal (Figure 10.4b).

50% Isolation Requirements and 50% Reachability Requirements

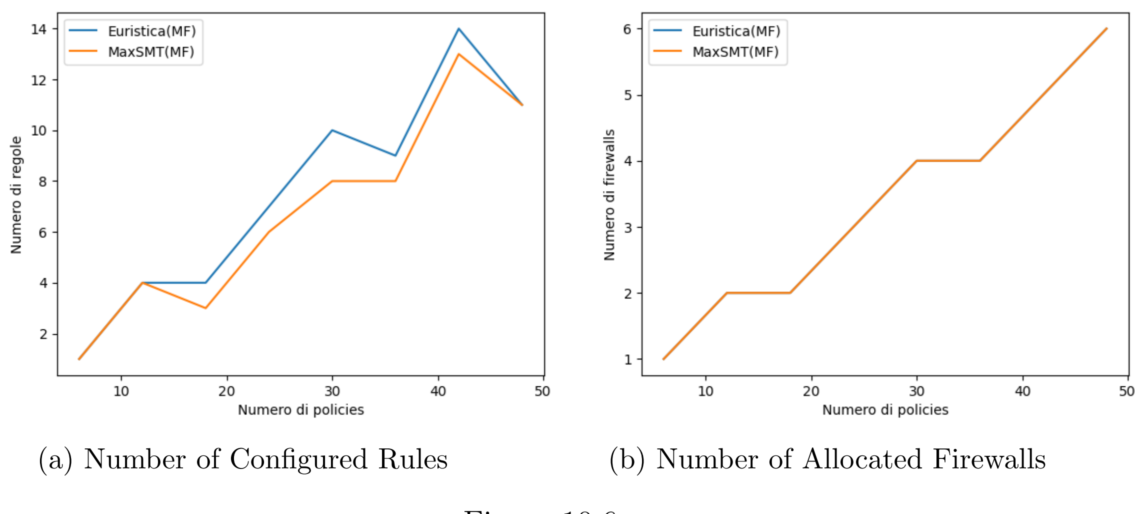


(a) Number of Configured Rules (b) Number of Allocated Firewalls

Figure 10.5

Just like in the *Geant* topology, in this scenario, the differences are more pronounced. The heuristic approach allocates a higher or equal number of rules compared to the MaxSMT approach, with an average difference of 23% more rules with a maximum difference of 66% (Figure 10.5a). However, the number of firewalls remains the same throughout (Figure 10.5b).

75% Isolation Requirements and 25% Reachability Requirements



(a) Number of Configured Rules (b) Number of Allocated Firewalls

Figure 10.6

Similar to the *Geant* topology, in this scenario, the differences between the heuristics and MaxSMT become less pronounced. Specifically, the heuristics allocates an equal number of rules to the MaxSMT or slightly more, with an average difference of 10% more rules (Figure 10.6a). Additionally, for firewalls, the heuristics allocates the same number of firewalls as the MaxSMT (Figure 10.6b).

10.1.3 Considerations

The previous optimality tests have essentially shown that, when considering the same traffic flows model, in this case the Maximal Flows, the solver achieves a higher level of optimality compared to the heuristics, as expected. Presumably, this result holds true for the other model as well, namely the Atomic Predicates. In particular, the largest difference between the solver and the heuristics occurs when the set of NSRs is comprised of half isolation requirements and half reachability requirements. In this scenario, the average difference in rules allocated by the heuristics is approximately 16% for the Geant topology and 23% for the Internet2 topology, with a maximum difference of 66% for both. However, when considering the other two cases (25% isolation and 75% reachability and vice versa), the differences are less pronounced, with an average difference of about 10% for both topologies. These differences between the heuristics and solver persist even after implementing the new post-processing algorithm for rules. Without this algorithm, the difference would have been much higher. In fact, as mentioned in Chapter 9, the

algorithm has reduced the number of rules configured by the heuristics by approximately 66% in the case of mixed requirements and 100% in the case of isolation requirements only.

10.2 Time and Memory Limits in the Heuristic Approach

This final phase of testing aimed to assess the maximum thresholds that the heuristic approach can reach before exhausting the available memory in the Eclipse IDE for Java. The testing included the following topologies:

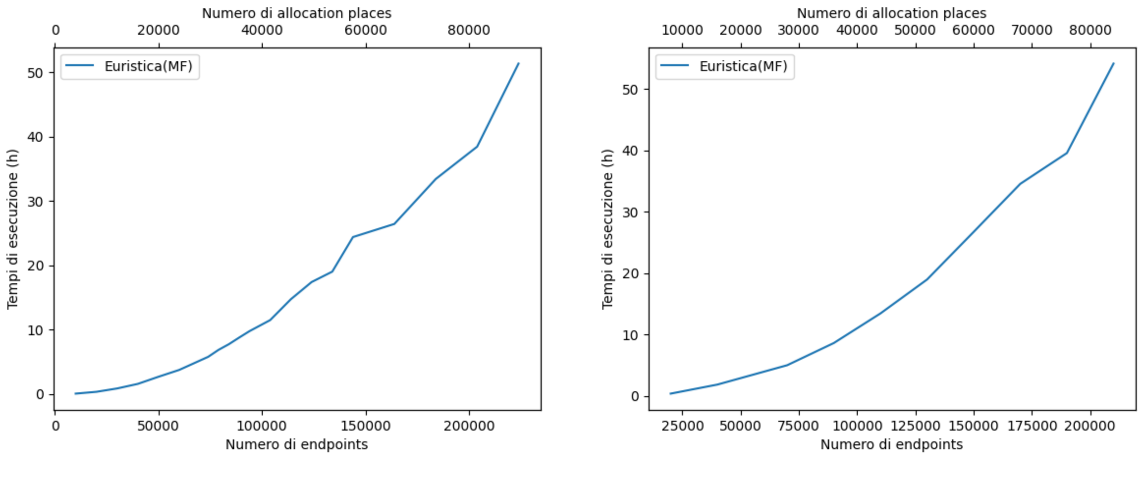
1. *Geant*
2. *Internet2*

The following two scenarios were tested:

1. 100% Isolation Requirements
2. 50% Rsolation Requirements and 50% Reachability Requirements

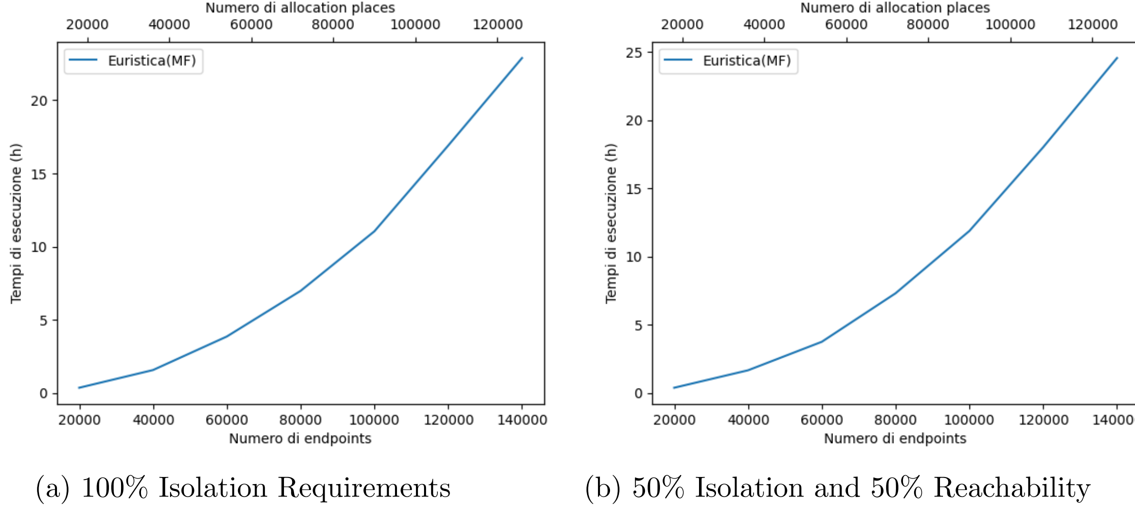
During each iteration, the number of endpoints was increased by 2000, while maintaining a ratio of 1:3 between the number of endpoints and the number of NSRs.

10.2.1 Geant Topology



(a) 100% Isolation Requirements (b) 50% Isolation and 50% Reachability

Figure 10.7: Execution Time Limits



(a) 100% Isolation Requirements (b) 50% Isolation and 50% Reachability

Figure 10.8: Execution Time Limits

10.2.2 Internet2 Topology

10.2.3 Considerations

From Figure 10.7, it can be observed that, considering the *Geant* topology, the heuristics reaches its limit with networks consisting of more than 200,000 endpoints, 90,000 allocation places, and approximately 600,000 NSRs, with an execution time of about 50 hours. These limits are reached in both scenarios, whether only isolation requirements are considered or when half of the requirements are isolation and half are reachability. The only difference is that the first scenario has slightly higher execution times with networks consisting of around 150,000 endpoints. This confirms that the heuristics is not affected by the added complexity of reachability requirements.

On the other hand, Figure 10.8 shows the results obtained for the *Internet2* topology. In this case, the limit is reached with networks consisting of approximately 140,000 endpoints, 130,000 allocation places, and 126,000 NSRs, with an execution time of about 25 hours. Again, there are no significant differences between the two scenarios. The only noteworthy point is that in the scenario with mixed requirements, the heuristics takes slightly longer execution times with large networks.

Regarding the differences between the two topologies, it can be noted that the *Internet2* topology reaches its limits with networks formed by about 100,000 fewer endpoints. This can be explained by the fact that, even with the same number of endpoints, the *Internet2* topology has a greater number of Allocation Places compared to the *Geant* topology, which implies more work for the heuristic algorithm as explained in Chapter 5. In fact, the *Internet2* topology reaches its limit with a smaller network in terms of endpoint quantity but larger in terms of Allocation Places (about 30,000 more than the *Geant* topology).

Chapter 11

Conclusions

VEREFOO (VErified REFinement and Optimized Orchestration) is a Java-based framework specifically designed to automate the process of determining the optimal placement and configuration of network security mechanisms within a virtualized network. This framework proves to be particularly valuable in managing complex networks where manual configuration is challenging and prone to human errors and misconfigurations.

The initial version of the system uses the Z3 theorem solver to solve a MaxSMT problem, ensuring correctness-by-construction and achieving optimality in terms of allocated firewalls and configured rules. However, scalability issues arose with large networks due to the NP-hard nature of the problem. To address this, a second version was developed using a heuristic algorithm, offering faster resolutions but potentially less precise results compared to the MaxSMT approach. This version strategically balances optimization, completeness, accuracy, and execution speed, striking a trade-off between these factors.

This thesis work aimed to test, evaluate and extend the new heuristic approach and compare it with the framework version based on the MaxSMT problem to understand their differences. While the heuristic algorithm is more efficient in terms of execution time, it allocates more firewalls and configured rules compared to the MaxSMT-based solution. Scalability and optimality tests were conducted on real-world topologies, and differences between Atomic Predicates and Maximal Flows in traffic modeling were also evaluated.

Firstly, the architecture of VEREFOO was described, briefly analyzing the functioning of its modules. Then, the two approaches considered in this thesis for traffic modeling, namely Atomic Predicates and Maximal Flows, were briefly described. Additionally, an overview of what a MaxSMT problem is, how it is modeled, and how it can be solved was provided. Subsequently, the new heuristic algorithm was described in detail, conducting an in-depth analysis of its operational phases.

Before beginning the testing campaigns, preliminary operations were conducted to enhance the subsequent tests. Specifically, some corrections were made to the heuristics code, and the two new topologies *Geant* and *Internet2*, inspired to real-world topologies, were designed and developed along with their Java-based generators. Additionally, operations were carried out to automate the classes used for

conducting tests, and a new random security requirements generator was developed, including port management in the various generated requirements.

Subsequently, an initial testing campaign was conducted to broadly illustrate the differences between the MaxSMT-based version and the heuristic-based version, as well as between the model using Atomic Predicates and the one utilizing Maximal Flows. The results revealed that the heuristic approach has inferior execution times compared to the solver and is also much more scalable, capable of handling larger networks and a greater number of Network Security Requirements. However, it falls short in terms of the number of configured rules. The number of allocated firewalls, on the other hand, remains similar. As a result, in a subsequent phase, the heuristic algorithm was enriched by implementing a post-processing algorithm, aiming to optimize the number of configured rules. Furthermore, the tests have shown that Maximal Flows are more efficient than Atomic Predicates in the heuristics, but instead are less efficient when considering the MaxSMT problem. Therefore, for subsequent tests, the version based on the heuristics and Atomic Predicates and the one based on MaxSMT and Maximal Flows have been discarded.

Finally, a second and third testing phase were conducted. The primary objective of the first phase was to demonstrate the efficiency of the new post-processing algorithm, while the second phase focused primarily on optimality and the number of configured rules. The number of rules remains lower in the MaxSMT-based version, but the differences with the heuristic approach have significantly decreased compared to before.

Potential future work could involve implementing a new version of the heuristic algorithm, such as the Branch-and-Bound method, and comparing it with the current one to assess if it achieves better performance. Additionally, it could be beneficial to evaluate the heuristic framework's behavior by incorporating other functions such as NAT (Network Address Translation) or Load Balancers, which are already implemented in VEREFOO. Finally, it could be interesting to develop a partial heuristic algorithm.

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)*, 2019, pp. 1–7.

[2] W. Xia, Y. Wen, C. H. Foh, D. Niyato, and H. Xie, “A survey on software-defined networking,” *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 27–51, 2015.

[3] “Open networking foundation,” https://opennetworking.org/.

[4] “European telecommunications standards institute,” https://www.etsi.org/.

[5] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, “Firmato: A novel firewall management toolkit,” *ACM Transactions on Computer Systems (TOCS)*, vol. 22, no. 4, pp. 381–420, 2004.

[6] J. D. Guttman, “Filtering postures: Local enforcement for global policies,” in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*. IEEE, 1997, pp. 120–129.

[7] N. B. Y. B. Souayeh and A. Bouhoula, “A fully automatic approach for fixing firewall misconfigurations,” in *2011 IEEE 11th International Conference on Computer and Information Technology*. IEEE, 2011, pp. 461–466.

[8] M. A. Rahman and E. Al-Shaer, “Automated synthesis of distributed network access controls: A formal framework with refinement,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 2, pp. 416–430, 2016.

[9] S. Bussa, R. Sisto, and F. Valenza, “Security automation using traffic flow modeling,” pp. 486–491, 2022.

[10] 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.

[11] H. Yang and S. S. Lam, “Real-time verification of network properties using atomic predicates,” in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, 2013, pp. 1–11.

[12] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, “Automated firewall configuration in virtual networks,” *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, 2023.

[13] ——, “Automated optimal firewall orchestration and configuration in virtualized networks,” in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–7.

[14] “Z3,” https://www.microsoft.com/en-us/research/project/z3-3/.

[15] Wikipedia contributors, “Heuristic — Wikipedia, the free encyclopedia,” https://en.wikipedia.org/w/index.php?title=Heuristic&oldid=1193914257, 2024.
[16] “Verefoo,” https://github.com/netgroup-polito/verefoo.
[17] “Geant,” https://network.geant.org/.
[18] “Internet2,” https://internet2.edu/.