Politecnico di Torino

Master of Science in Computer Engineering

Master Degree Thesis

Analysis and Enhancement of the Automatic VPNs Configuration

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

Candidate
Gianmarco Bachiorrini

Academic Year 2023-2024

Summary

In today's rapidly evolving Information Technology landscape, automation is a key objective across all sectors, including cybersecurity. The security of a system can be completely compromised by even the minor of configuration errors, highlighting the importance of minimizing human intervention in security configurations. Manual configuration of security systems, such as VPNs and packet filters, is not only prone to errors but also likely to result in suboptimal solutions: studies have shown that even seasoned security experts can introduce anomalies and errors when configuring complex systems manually. Therefore, automating such crucial tasks would greatly improve the overall quality and effectiveness of a network's security, while minimizing the attack surface exploitable by potential attackers.

The VEREFOO (VErified REFinement and Optimized Orchestrator) framework perfectly fits the role of a tool capable of automating the allocation and configuration of security systems, producing as output network configurations that are both correct and optimal. VEREFOO is a framework developed at Politecnico di Torino which lays its foundation on the formal methods approach, whose core design allows the achievement of three important goals: firstly, ensuring formal correctnes of the produced solution by construction; secondly, optimization by selecting the most efficient option in the solution space; finally, as a consequence of the first two, eliminating the need for post-verification.

This thesis aims to analyze the problem of automating the allocation and configuration of VPNs from both conceptual and practical perspectives, and enhance the implementation of the approach in VEREFOO. In particular, the thesis focuses on the analysis of how VEREFOO manages the formulation of the VPNs' configuration process as a Maximum Satisfiability Modulo Theories (MaxSMT) problem: starting from the generation of the network flows and constraints applied to the Network Security Functions (NSF) allocated on the graph, the analysis examines every computational step that ultimately leads to the production of the final outcome.

Through careful code reviews and debugging, the objective is to detect and fix the issues affecting the current version of the framework, while also studying new optimization strategies to enhance the tool's performance. The analysis revealed several defects in the framework's code, such as inefficient generation of constraints, an incorrect assignment of weights and grouping of constraints, imprecise generation of the various flow path and a redundancy issue affecting the quality of the output solution. Additionally, for not-yet resolved challenges like the achievement of feasible performances when managing large-sized networks, various strategies and approaches are proposed. The intrinsic complexity of the VPNs

auto-configuration problem leaves very little room for sub-optimizations, especially related to the generation of constraints, as they can lead to unexpected and ambiguous output scenarios.

Finally, an evaluation of the current framework's implementation is presented, in order to assess its strengths and weaknesses. The evaluation offers insights into areas where VEREFOO, and more broadly, the state of the art of VPNs automatic configuration, can advance in the future.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Thesis introduction

The nowadays landscape of security management for virtualized networks has been as turbulent as ever. As attackers grow more skilled and aware of potential vulnerabilities to exploit, system administrators must keep up the pace with the evolving nature of virtualized networks, managing increasingly complex topologies and security configurations. The configuration of complex security systems, such as NAT, IDS, firewalls and VPNs, can become particularly time-consuming, even when limited to small-medium sized business networks. When we factor in the high risk of misconfiguring these critical systems, the balance shifts dangerously in favor of the attackers.

From this perspective, the cybersecurity automation increasingly appears to be a necessity rather than merely an innovative branch of the cybersecurity field, as we aim to build high-quality security solution to protect our assets. Cybersecurity automation, particularly in the configuration of security systems, is a relatively new and still underexplored approach to the setup and enforcement of security rules in a network,which are still largely managed manually in most cases.

With its foundation firmly rooted in this philosophy, the VEREFOO (VErified REFinement and Optimized Orchestrator) framework has been designed and developed, which purpose is to automatically allocate and configure Network Security Functions (NSF) on the network. VEREFOO's strength lies in the combination of security automation with a formal methods based approach. This approach formulates the configuration and allocation of NSFs as a Maximum Satisfiability Modulo Theories (MaxSMT) problem, with the framework's embedded SMT solver automatically computing its resolution. This means that the security administrator's only task is to provide the framework with a Service Graph (SG) on which the NSFs will be deployed, along with the set of Communication Protection Policies (CPPs) they wish to enforce within the network, while the framework automatically generates an output configuration, guaranteed to be both correct and optimal.

In the literature, VEREFOO has proved the effectiveness of its approach, especially in the packet filters configuration capability, as the works [1] [2] [3] exhaustevely analyze the model, while [4] presents a practical demonstration of the

framework in action. Building on its excellent performance in configuring firewalls, VEREFOO has been extended to support the allocation and configuration of Virtual Private Networks (VPNs), a mechanism that is becoming increasingly relevant. However, VPNs are higly complex security systems, whose configuration involves a greater number of factors to consider with respect to other security systems, and currently the framework is not yet capable to achieve the same computational performances as it does when configuring firewalls.

The objective of this thesis is to analyze VEREFOO's current implementation for ensuring secure communications in a network through VPNs and to enhance it by introducing new pruning and optimization strategies, while also identifying areas for potential improvement and fixing minor issues.

1.2 Thesis description

The thesis is organized as follows:

- **Chapter 2**: This chapter provides a background on cybersecurity automation, discussing its benefits and challenges, and explores the current state of the art. It also delves into VPN architecture, covering the key concepts, components, and protocols relevant to VPNs. Finally, it presents an overview of the VEREFOO framework, setting the foundation for the thesis.
- **Chapter 3**: This chapter outlines the thesis's objectives, focusing on the automation of VPN configurations using the VEREFOO framework. It details the steps for analyzing and enhancing VEREFOO, including identifying issues and implementing optimization strategies for VPN configuration.
- **Chapter 4**: This chapter presents an in-depth analysis of VEREFOO's capabilities for automatic VPN configuration. It describes the framework's architecture, focusing on constraint generation and detailing the algorithms used to allocate and configure VPN components.
- **Chapter 5**: This chapter addresses the identified issues and limitations within VEREFOO, discussing enhancements made to improve the framework's performance. It includes the development of new optimization strategies and their implementation to resolve inefficiencies in VPN configuration.
- **Chapter 6**: This concluding chapter summarizes the main findings and contributions of the thesis and suggests areas for future research to further advance the automation of VPN configuration.

Chapter 2

Background

This chapter provides a general overview of the background and context on which the thesis is based.

In the first section, the motivation of introducing automation in the cybersecurity field will be presented, highlighting its benefits while also exponing the limitation of traditional manual network security management. Then, the focus shifts on the state of art of cybersecurity automation, analyzing which concepts related to this topic has been covered by the literature and which, instead, are still unexplored. The aim is to provice a clear picture of what the literature has focused on.

The second section of this chapter will analyze the VPN architecture. This analysis comprehend a deep focus on its purpose, its characteristics and its functioning. This way, the key features and concepts that need to be formalized later in the VEREFOO framework will be highlighted.

Finally, a brief overview of VEREFOO will be provided, covering the inputs, outputs, methodology, and models that are relevant for the subsequent in-depth analysis.

2.1 State of the art of cybersecurity automation

2.1.1 Motivation for shifting towards security automation

Managing network security requires the security administrator to analyze the computer network to identify assets that need protection and potential threats, draft the necessary security requirements to achieve the desired security level, select the appropriate NSFs to enforce these requirements, and finally allocate and configure the NSFs accordingly. For instance, if a security requirement specifies that all requests originating from a known malicious address must be rejected, the security administrator would identify a packet filter as the appropriate NSF to provide this protection; consequently, a firewall would be allocated and configured with the necessary filtering rule to enforce this requirement. Although it may initially seem like a straightforward task involving a linear sequence of actions, the complexity of

management can quickly escalate when multiple NSFs are required, particularly as the network size and the number of security requirements to be enforced increase. This often leads to solutions that unintentionally introduce one or more anomalies, which are inconsistencies caused by the security administrator during the NSF configuration phase. Following the classification process performed in [5], anomalies can belong to one of the following categories:

- Conflict: this type of anomaly happens when the enforcement of a security rule is incompatible with the enforcement of another. The conflicting security rules could be enforced by the same NSF, or by different NSFs (even of different types, like a VPN gateway and a NAT). A simple example is two filtering rules with identical conditions but contradictory actions. More subtle conflicts can arise, for instance, when a NAT and a VPN gateway are misconfigured, causing the NAT to modify the data packet's IP address so that it no longer matches the VPN's security association. Generally, conflict-type anomalies lead to the failure to apply protection and often result in network connectivity issues as well.

- Error: this type of anomaly occurs in presence of an hard misconfiguration. For example, a VPN gateway may be expected to create a TLS tunnel, but it fails to do so because the necessary TLS module hasn't been installed. More broadly, the security administrator's solution design might depend on a node to enforce a security rule, but the node lacks the necessary capability to implement that rule. In most cases, this type of anomaly causes a non-enforceability situation and can potentially prevent any data exchange.

- Sub-optimization: This type of anomaly occurs when the network configuration reduces throughput or results in wasted resources. In the context of firewall configuration, a sub-optimization anomaly can occur when filtering rules that could be combined are left separate, causing certain rules to never be triggered and ultimately leading to wasted memory. In the case of VPNs, various sub-optimizations can occur, with one of the most impactful being the allocation of an excessive number of VPN gateways. This leads to a significant reduction in overall network throughput as unnecessary traffic is subjected to computationally intensive cryptographic operations. Less impactful sub-optimizations include redundant tunnels that encapsulate already secure tunnels, adding unnecessary overhead on nodes, or configurations that force data to travel unnecessarily long paths, resulting in suboptimal routing. In a cloud computing environment, suboptimal assignment of Virtual Machines to nodes can lead to increased energy consumption and network congestion, as addressed in research such as [6]. While these types of anomalies may not prevent achieving the desired security level, resolving them is crucial for enhancing network performance and reducing the vulnerability of the security configuration to DoS attacks.

The presented classification of anomalies is not intended to be exhaustive, as the literature offers thorough analyses of anomalies specific to each major security domain. For example, [7] examines anomalies arising from enforcing security policies through VPN gateways, while [8] addresses anomalies related to filtering rules.

Here, the aim is to highlight the typical scenarios in which a security manager might inadvertently introduce critical errors, conflicts, and sub-optimizations, which are highly common when configuring such complex security systems manually. In addition to offering the aforementioned taxonomy, the study [7] conducted an empirical assessment to analyze the relationship between the number of introduced anomalies and the expertise level of the administrator enforcing the Communication Protection Policies (CPP). The analysis revealed that 93% of administrators involved in the experiment introduced at least one anomaly. As expected, solutions designed by less experienced professionals were more likely to generate impactful inconsistencies, such as conflicts and errors, with 70% and 60% respectively occurring among administrators labeled as having "low expertise." Similar percentages were observed for those with a medium level of expertise. However, the more interesting—and concerning—aspect is that even expert professionals introduced a considerable number of anomalies, particularly within the sub-optimization category. The research indicates that while their advanced expertise and deeper understanding of the network generally prevent them from introducing conflict anomalies, this same expertise can also lead to the design of suboptimal implementations. In summary, the empirical assessment demonstrated that these anomalies frequently occur in real-world scenarios, regardless of the security administrator's level of expertise.

Adding to the challenges faced by security managers who manually perform these complex tasks, the 2024 State of Security Report by Splunk [9] identifies the misconfiguration of security systems as both the most common threat vector (38%) and the most concerning threat vector (35%). Furthermore, the 2024 Data Breach Investigation Report by Verizon [10] highlights that misconfiguration remains one of the most frequent causes of data breaches, with the healthcare industry being the most affected. The fact that the healthcare industry is the most affected suggests that, despite the high value and critical importance of protecting its data assets, the expertise of security managers responsible for configuration may often be insufficient, possibly indicating a lower investment in safeguarding patient data. Moreover, to complete the picture, it is of foundamental importance the consideration of the costs of the breaches. In this regard, the IBM's 2024 Cost of Data Breach Report [11] states that the global average cost of a data breach increased 10% over the previous year, reaching an astonishing amount of $4.88M. In particular, the costs of breaches caused by system errors and cloud misconfiguration averaged respectively $4.07M and $3.98M. Unsurprisingly, the healthcare industry remains the most costly sector for data breaches, with an average expense of $9.77 million per incident. These statistics reached skyhigh numbers, highlighting the enormous economic impact a single breach caused by a configuration anomaly can generate.

Having discussed the real world scenarios where the manual configuration can lead to anomalies and how severe their consequences can be from both technical and economic points of view, it is equally important to highlights the benefits and advantages of moving towards the automation of the tasks.

"Automation is the application of technology, programs, robotics or processes to achieve outcomes with minimal human input" is the definition provided by IBM to synthetically describe what automation is. However, as noted in [12], the term "automation" was coined as early as 1936, originally applied within the context

of industrial production, referring to the transfer of workpieces between machines without the need for human intervention. Since 1946, when the Automatic Department of Ford Motor Company was firstly established, the focus on this science has increased exponentially. Today, nearly every sector has been transformed by automation, with Agriculture, Construction, Energy, Healthcare, Transportation, and Manufacturing being just a few of the many areas that have benefited from its advancements. Regardless of the context in which it is applied, the common denominator of automation remains consistent: it aims to minimize the number of tasks that need to be performed manually by human personnel. This principle holds true in the realm of security configuration and orchestration as well. The main focus of automation, as far as network security is concerned, is on the NSFs allocation and configuration tasks. Automated security configuration and orchestration can be accomplished through various approaches, including Artificial Intelligence-enabled techniques (such as Machine Learning, Deep Learning, and Reinforcement Learning) and formal methods-based approaches. In either case, the benefits can be summarized as follows:

- Minimized Human Intervention: with automated security configuration, human administrators are no longer required to design security solutions from scratch. Their tasks are instead limited to defining the security properties that need to be enforced on the network and providing assistance and maintenance for the automated system. Shifting their focus primarily to monitoring activities can greatly enhance efficiency, accuracy, and responsiveness, enabling administrators to manage multiple systems concurrently while significantly reducing the likelihood of human errors.

- Lower level of expertise required: since the security management operations are automated, the high likehood of conflict and error anomalies generated by solution designed by less experienced system administrator would no longer be an issue. This approach allows smaller companies, which may not have the resources to hire highly experienced professionals due to their high salary demands, to still aim for high levels of security in their networks.

- Better management of large and heterogeneous networks: the trend on the raise of larger and heterogeneous network is yet another issue that furtherly complicates the traditional manual security management. However, an automated tool can easily abstract different implementations of the same NSF and subsequently perform, for each individual device, an efficient translation to suit its specifics. At the same time, improved performances in managing large-scale computer network is guaranteed, thanks to the comprehensive overview of the entire network architecture that humans cannot naturally achieve.

- Optimization: as observed in [7], the sub-optimization type of anomaly was the most common one across all expertise level when security management is performed manually. With an automated approach, instead, not only optimization can be achieved more easily, but it would be possible to implement different optimization profiles and then let the security administrator deciding which one is more suitable to his needs. For example, when configuring

VPNs, the administrator could choose to prioritize the generation of end-to-end VPNs, resulting in less VPN gateways allocated on the network, or, on the contrary, the administrator could prioritize the performance of network communication in terms of bandwidth, so prefering site-to-site VPNs.

Although automation has historically proven to be a game-changing innovation in various fields, it is often met with justified skepticism when introduced into a new domain. The decision to transfer control to software or hardware is always approached with caution, particularly when the automation involves critical tasks like IT security. The papers [13] and [14] discuss respectively what is the general perception of the automation of tasks traditionally performed by humans and how automation in cybersecurity could cause more harm than good. However, thanks to the IBM 2024 Cost of a Data Breach Report, we know that more than 67% of organizations is at least using AI and automation on a limited basis, and the 31% use them extensively, representing a 10.7% increase in use. This shows that the trust in these automated tools is slowly growing, confirmed by the 44% of respondents interviewed by Splunk citing AI as among their three main initiatives in 2024, now surpassing cloud security. Moreover, almost 90% of them believe that it can help organizations hire more entry-level talent since AI and automation can be useful to develop their skills once they are hired, and 65% think that it will also allow seadoned security pros to be more productive. To conclude the overview of the nowadays perception and impact of automation, the organizations extensively using these tools saved averagely $1.88M when a breach occured, compared to the ones deciding to not rely on AI and automation.

2.1.2 Analysis of the state of the art

The literature review performed in this thesis focused on the articles, publications and books related to the automatic configuration of firewalls and VPNs in order to better understand which objectives have been achieved, and which objective are still pursued. The search process was mainly performed in the Google Scholar web engine and in the IEEE Digital library database, using the following search strings:

- *((firewall OR (packet AND filter)) AND (automation OR automatic OR automated OR programmability OR programmable) AND (configuration OR configured))*
- *((vpn OR (virtual AND private AND network)) AND (automation OR automatic OR automated OR programmability OR programmable) AND (configuration OR configured))*

Automatic Firewall Configuration

Among the papers reviewed on the subject of automatic firewall configuration, [15] is the earliest to propose a tool designed to assist security administrators by automatically generating and analyzing firewall configurations across a network. The configuration algorithm requires three key inputs: the network topology, a set of organizational firewall policies that define permissible traffic for each machine, and

trust assumptions that indicate whether a node is considered trusted. Once the feasibility of the desired policy is established, the automated configuration generator proceeds through an iterative process. However, the proposed tool has several limitations. It primarily focuses on preventing spoofed traffic, which consists of packets that falsely claim a different origin than their actual source. While spoofed traffic is indeed a significant threat, firewalls are deployed for a broader range of purposes, making the tool's functionality somewhat specific rather than general. Additionally, it is limited to filtering inbound traffic, excluding the filtering of outbound traffic. The algorithm also fails to fully address cases where destination nodes or trusted nodes lack the capacity to filter traffic. Furthermore, the tool only supports whitelisting policies and does not include any optimization objectives. Despite these limitations, the tool exhibits notable strengths, including correctness by construction and its applicability within distributed network architectures, which demonstrate its potential for further development and refinement.

A similar achievement is presented in [16], though through a fundamentally different approach. The framework described in this study introduces a low-level language based on the IETF policy framework, capable of managing various types of firewalls, including packet filter firewalls and application layer firewalls. It also includes a tool for analyzing potential inconsistencies in firewall policies prior to deployment. The static analysis, grounded in formal specifications, aids in identifying common errors such as shadowing, correlation, generalization, and redundancy in firewall rules. After policy deployment, the framework automatically validates enforcement by generating test packets and monitoring the firewall's response, producing a detailed report for the administrator. While the ability to manage different firewall types, combined with pre-deployment static analysis and post-deployment runtime validation, represents significant innovations, the framework has certain limitations. First, a high-level language for policy specification is generally preferred over low-level languages, as it allows security requirements to be expressed in a more abstract and manageable manner. Second, the framework lacks any consideration of optimization, and no information is provided regarding its performance and scalability. At the time of publication, only the policy analysis tool had been fully implemented, with the enforcement testing functionality still under development.

An intriguing approach to the problem of automated firewall configuration is presented in [17], where the authors adopt a procedural method that aims to minimize security risks while accounting for connectivity requirements, usability, and budget constraints. The proposed synthesis process is capable of managing a distributed filtering architecture by generating rule configurations for each firewall to satisfy both risk and connectivity requirements. Additionally, it aims at optimizing firewall placement within the network to reduce spurious traffic—traffic destined to be dropped by a firewall—which unnecessarily consumes network bandwidth and increases vulnerability to Denial of Service (DoS) attacks. However, to address the Distributed Firewall Synthesis Problem (DFSP), the authors had to rely on a greedy heuristic algorithm, since their integer programming solver could not compute the optimal solutions for larger networks. While this approach improves scalability, it sacrifices optimality, as is typical of heuristic methods. Although the deviation from the lower bound of the optimal solution remains within acceptable

limits, true optimality is still not guaranteed.

The application of formal argumentation and preference reasoning for the analysis and automated generation of firewall policies is explored in [18]. A notable advantage of this approach is its declarative nature, as the logic-based, declarative specification of firewall requirements and configuration properties enhances the comprehensibility of the policy set for administrators. The validity of this method has been demonstrated through its application to a real-world firewall configuration of moderate size (150 rules). However, the approach has several significant limitations. It does not address the configuration of multiple firewalls within the same network or the detection of inter-firewall anomalies; additionally, the challenge of optimizing firewall configurations is not addressed.

In [17], the authors address the research question by formulating the problem as a Boolean satisfiability problem, introducing a framework that leverages an SMT solver to identify the correct and optimal security configuration. The objective of their configuration synthesis problem is to maximize both usability—defined as the accumulated usability of a node based on all service flows to that node—and overall isolation within the network, while simultaneously satisfying various security requirements and organizational business constraints. A notable strength of the proposed framework is the broad range of NSFs it supports, including not only firewalls but also IDS, IPsec, and NAT. Furthermore, the framework demonstrates satisfactory performance despite its focus on optimization in terms of both security configuration and device placement.

Nevertheless, all the previously discussed works focus exclusively on traditional computer networks, which no longer align with current trends that increasingly rely on Software Defined Networks (SDNs) combined with the Network Functions Virtualization (NFV) paradigm. These technologies offer users and administrators greater flexibility and agility in network configuration. The study in [19] partially embraces these paradigms by proposing a semantic-based tool for firewall configuration, specifically targeting Netfilter, a firewall system integrated into the Linux kernel. Netfilter's behavior is highly dependent on the order of its rules, as rule sequencing directly impacts how the firewall functions. To address this, the authors introduce a formal model of the Linux firewall system, which abstracts its key concepts, and develop a simple yet powerful language that enables users to specify firewall configurations independently of rule order. However, the tool's limitation to Netfilter undermines its flexibility, restricting its broader applicability in diverse environments.

A network configuration synthesis tool that targets both traditional and virtual networks is proposed in [20]. The synthesis process described by the authors utilizes stratified Datalog to model the network's behavior, with routing requirements expressed as constraints, thereby reducing the input synthesis problem to the satisfiability of SMT constraints. To ensure network reliability and scalability, the approach relies heavily on distributed protocols to compute the forwarding state. While the goals of reliability and scalability are effectively achieved, the methodology lacks formal verification techniques applied to the synthesized configurations, nor does it employ a formal correctness-by-construction approach.

Finally, the group of papers ([20-23]) shifts the focus from the automatic generation and configuration of firewall policies to the analysis of existing configurations, aiming to detect and resolve misconfigurations and anomalies. Specifically, [21] proposes a method based on Firewall Decision Diagrams (FDDs) for analyzing firewall configurations. In [22], a formal approach is utilized to manage firewall misconfigurations, while [23] presents an automatic test generator based on white-box testing. Lastly, [24] introduces a framework capable of testing firewall implementations by generating both ACL policies and the corresponding packets for their enforcement.

Automatic VPN configuration

In this research area, one of the earliest works to recognize the significance of abstracting Virtual Private Networks (VPNs) is [25]. After identifying the limitations of manual configuration, the authors propose a VPN design aimed at satisfying the specifications provided by the customer, while simultaneously maximizing resource availability for the VPN Service Provider (VSP). Once the customer and the VSP establish a Service Level Agreement (SLA), the partial VPN specification must be translated into a complete specification. Upon achieving a full description, the VPN can be expressed as a set of "switchlet" specifications, which partition the physical resources to ensure that VPNs are isolated and do not interfere with one another. The problem of determining the optimal route for a VPN is compared to finding a minimum Steiner Tree, where the objective is to identify the least expensive subgraph based on a cost function. The cost function is automatically generated by parsing the SLA, which integrates customer requirements with the specific conditions of the VSP. Since finding the minimum Steiner Tree is an NP-complete problem, the task of determining the optimal VPN topology, as outlined in the paper, is also NP-complete. To bridge the gap between suboptimal and optimal solutions, the authors analyze a brute-force approach, constructing every possible connected subgraph that satisfies the VPN description, calculating the cost of each, and selecting the least expensive. As expected, this brute-force algorithm performs poorly in practice. However, the paper made significant contributions to the field of automated VPN configuration by (i) proposing a method to translate high-level customer requirements into low-level resource specifications, thereby enhancing usability and easing the administrative burden, and (ii) formulating the problem of optimal VPN topology in a way that balances customer needs with VSP resource constraints.

Another significant milestone in the evolution of research on automatic generation of security policies for VPNs is represented by [26]. In this paper, the authors make a critical distinction between security requirements and low-level policies. Security requirements reflect the overarching security objectives and are independent of specific implementations, while low-level policies are concrete actions designed to meet these objectives. This separation provides a clear definition of policy correctness, whereby a low-level policy is considered correct only if it satisfies all corresponding security requirements. The authors classify security requirements into four main categories: (1) Access Control Requirements, which restrict access to trusted traffic only; (2) Security Coverage Requirements, which mandate that security protections must encompass all links and nodes within a specified area; (3) Content Access Requirements, which specify that certain nodes (e.g., firewalls, IDS) must have access to the content of certain traffic, ensuring that the traffic is

not encrypted at those nodes; and (4) Security Association Requirements, which define the need for—or prohibition of—establishing Security Associations of certain security functions between specific nodes. To systematically generate policies that fulfill the specified security requirements, the paper introduces two algorithms. The first, known as the Bundle Approach, groups entire traffic flows into disjoint bundles based on the given requirements. For each bundle, a tunnel or a chain of tunnels is created to satisfy the Security Coverage Requirements (SCR) of the area. While this approach guarantees completeness and correctness, it is less efficient and scalable due to the computational overhead involved in bundling traffic, which requires extensive difference calculations. The second method is the Direct Approach, which generates policies directly corresponding to each SCR without bundling traffic into groups. Compared to the Bundle Approach, the Direct Approach generates fewer policies and requires less frequent updates, making it a more scalable and efficient solution, but also not complete, meaning that future work for optimization is required.

The same authors expanded their original work several times, proposing with [27] a new approach with the aim to reduce to the minimum the number of policies generated, while [28] address the challenge of distributed policy management through an ad-hoc framework.
In particular, [27] introduces the Ordered-Split Approach, an alternative method to the previously discussed Direct and Bundle approaches, aimed at addressing the redundancy and quality issues that affect them. This new algorithm is inspired by traditional task scheduling, where the authors draw an analogy between the start and end times of a task and the start and end points of a tunnel. The goal is to eliminate any overlaps between tunnels, ensuring that no two tunnels share the same start or end time, resulting in what is referred to as a Canonical Solution. The same logic is applied to security requirements, with the transformed set of requirements being termed Tie-Free Requirements Sets. These requirements are processed in a sorted order, and at each step, any existing tunnels that overlap with the current requirement are split. Additional tunnels needed to complete the chain for the current requirement are then added. A final cleanup phase ensures that the resulting solution remains canonical, after which the algorithm proceeds to process the next requirement. The Ordered-Split Approach demonstrates a significant improvement over previous methods, as the number of policies generated stabilizes once a certain threshold is reached, in contrast to the older approaches that cause the number of policies to increase dramatically as the number of requirements grows.
On the other hand, [28] encapsulates previous works within BANDS, a framework designed for inter-domain security policy management. The BANDS architecture employs a hybrid structure, integrating both centralized and distributed systems, and operates in two main phases: (1) Autonomous System (AS) route path discovery and (2) an inter-domain collaborative protocol for policy negotiations among the ASes discovered in the initial phase. Each AS within the framework contains a Requirement Server (RS), which is responsible for managing policy negotiations with other RSs in the distributed environment. When an RS receives a negotiation message from a remote RS, it computes the corresponding security policies using one of the automated policy generation algorithms discussed earlier. Once the policies are determined, each RS communicates the necessary security policies

to both local and remote routers, establishing the required Security Associations (SAs) for a specific traffic flow. This framework enables effective cooperation and policy enforcement across multiple domains, facilitating more comprehensive security management in complex, inter-domain network environments.

However, very few studies in the literature propose solutions to the problem of automatic configuration of VPNs in a non-traditional network. The paper [29] proposes a method that integrates SDN with IPsec to streamline the configuration and management of IPsec VPNs by using OpenFlow switches and controllers. The architecture consists of an SDN controller, an OpenFlow switch and a VPN server, where the switch acts as the IPsec VPN client, while the controller issues configuration parameters to the client through the IPsec configuration module equipped on the SDN controller. In this setup, the SDN controller manages the flow of data packet, enforces IPsec policies and ensures that only authenticated devides can establish VPN connections; the authentication process relies on the RADIUS architecture.

The work in [30] adopts a similar philosophy, presenting a solution for managing IPsec Security Associations (SAs) using SDNs, with the novel distinction of handling two cases: IKE and IKE-less. In the IKE case, the complexity of IPsec management is shared between the network resource and the SDN controller. The network resource implements IKEv2 to manage the IPsec SA, while the SDN controller provides the necessary configuration for IKEv2 to function. In contrast, the IKE-less case simplifies the network resource by incorporating only the IPsec logic, while delegating all key management operations to the SDN controller, which is capable of independently deriving the IPsec SAs. This approach reduces the complexity at the network resource level, centralizing key management within the SDN controller. The architecture is defined through three key interfaces: (1) the Northbound interface allows administrators to configure the SDN controller with Flow-based Protection Policies (FPP) using a high-level language to describe how data flows should be managed between networks under its control; (2) the Southbound interface manages the interaction between the SDN controller and the network resource, enabling the controller to provide IPsec configurations to the network resource; (3) the East/West interface operates between two SDN controllers and is required when network resources managed by different controllers are involved in the IPsec protection of a specific flow.

Finally, a different approach is proposed by [31], where the concepts of DevOps are applied to the resolution of the problem of automated VPN configuration in an SDN-environment. This approach emphasizes enhanced usability and manageability by enabling user interaction through a graphical user interface (GUI). The SDN Hybrid model allows equipment from different providers to be connected through VPN tunnels configured automatically. The architecture consists of three planes: (1) the Application plane, which provides the web-based GUI for users to input the necessary details for VPN tunnel deployment; (2) the Control plane, which defines the operational model; and (3) the Data plane, which includes the physical infrastructure required to support network traffic. Through the GUI, the user provides both the IKE parameters (encryption algorithm, hashing algorithm, Diffie-Hellman group), which are used to set up a secure authenticated connection between the two

concerned router, and the infromation regarding the IPsec (protocol and other relevant details), which are used to prepare the mandatory SAs to establish the VPN tunnel.

Final considerations and open issues

On the basis of this extensive analysis, state-of-the-art studies about automatic NSF configuration have several limitations that should still be addressed.

In terms of firewall configuration, one of the most evident shortcoming in the literature is the lack of focus on optimization, which is often overlooked entirely. Optimization is crucial not only for enhancing network performance but, more importantly, for elevating the overall quality of the security infrastructure. Furthermore, even when optimization is not the primary goal, the scalability of the approaches analyzed rarely reaches the levels required to manage medium-to-large real-world network scenarios. Another common shortcoming in much of the existing research is the limited applicability to distributed network architectures. From a security perspective, distributed architectures are significantly preferred over centralized ones, as the latter introduce a single point of failure, increasing the risk to the entire network.

On the VPN configuration side, the situation is similarly lacking: heuristic approaches are required to make current methods feasible for modern network sizes and to handle the increasing number of security requirements needed to deploy a robust and reliable VPN infrastructure. Additionally, none of the studies reviewed provide support for TLS technology, despite its rapidly growing relevance. Moreover, with a few rare exceptions, formal methods are generally overlooked. This absence deprives the proposed solutions of the soundness and stability that formal verification or correctness-by-construction can offer.

To sum up, the literature is still in search of an approach that combines correctness, optimality, and scalability, while addressing both traditional and virtual networks, as well as distributed architectures.

2.2 VPNs Architecture

2.2.1 Definition

> A VPN is a communications environment in which access is controlled to permit peer connections only within a defined community of interest, and is constructed though some form of partitioning of a common underlying communications medium, where this underlying communications medium provides services to the network on a non-exclusive basis.

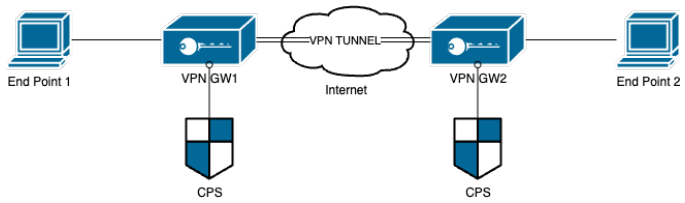
This is the formal definition provided by [32]. In other words, a Virtual Private Network (VPN) is a networking architecture implemented over a public network (such as the internet), with the purpose of creating a secure and virtual connection between peers. Two terms are worth a deeper explanation, *virtual* and *secure*. In

this context, the adjective "virtual" refers to the fact that no dedicated physical infrastructure or exclusive channel is being created between the peers, but rather that the underlying public network is shared by other organizations and users that are unaware of the private channel established by the peers, making the communication appear as though it is happening over a private, isolated network. The other key term, "secure", means that the virtual connection is designed to uphold specific security properties that provide a degree of protection against unauthorized actors who are not part of the communication: certain security properties, such as authentication, are mandatory in any VPN implementation to ensure that only authorized peers can establish a connection; other properties, like confidentiality through encryption, are optional and depend on the specific requirements of the peers.

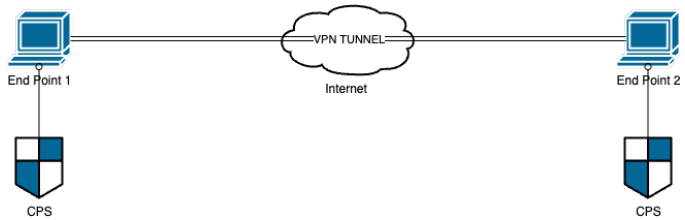
2.2.2 Components and Actors

In a traditional VPN architecture, we can distinguish the following core actors:

- Communication Protection Systems: systems located at the VPN's borders that actively apply protection on the VPN's traffic through a set of mechanisms and protocols. They can be placed directly on the End Points (if they support the required VPN protocols and ciphersuites) or on the VPN Gateways, depending on the VPN topology and purpose.
- End Points: network nodes or subnetworks representing the users or systems that need to securely connect to a remote network or service through the VPN.
- VPN Gateways: network nodes specifically designed to host the CPS and create the VPN tunnels.



(a) CPS equipped on gateways



(b) CPS equipped on end points

Figure 2.1: Example of VPN architecture

As shown in Figure 2.1, the allocation of CPS determines where the VPN tunnels start and end. In the first scenario, the CPSs are allocated on the VPN gateways,

which are responsible for establishing the tunnel and applying the necessary security measures to the traffic. In the second scenario, the traffic is protected directly from End Point 1, eliminating the need for a VPN gateway, as the tunnel is established directly between the peers. Many more hybrid models exist, and their differences will be explored in greater detail in the subsequent subsections of this chapter.

The previous figure represents a highly simplified VPN architecture. In real-world scenarios, numerous additional network functions and services, such as firewalls, Network Address Translation (NAT), Intrusion Detection Systems (IDS), and others, enrich and complicate the network topology. Furthermore, the design of a VPN may vary significantly depending on the specific context, such as whether it is deployed in a commercial or business environment, each of which may have different security and operational requirements.

2.2.3 VPN Tunnels

In networking, a tunnel is a communication channel that encapsulates data packets of one protocol inside another packet of the same or of a different protocol as its payload and adds a new header before sending it, allowing communication between two entities over an intermediary network. The new headers contain specific routing information necessary for transmitting the packet over the intermediary network, typically the Internet. Once the packet reaches its final destination, it is decapsulated, allowing the recipient to access the original packet and its contents.

In the context of VPNs, to establish a secure tunnel between communicating entities, packets are protected prior to encapsulation. This protection involves the calculation of a Message Authentication Code (MAC) to ensure integrity and authentication, as well as encryption to guarantee confidentiality. When strong cryptographic algorithms are employed, the only viable attack on the tunnel would be to disrupt communication entirely.

We can distinguish three main VPN topologies which rely on different tunnel edges placement:

- End-to-End (E2E): In this VPN topology, protection on network traffic is applied and removed directly by the communication endpoints. It is well-suited for use cases that prioritize direct and secure communication, such as privacy-focused applications (e.g., VoIP services) and Internet of Things (IoT) devices.
- Site-to-Site (S2S): In this VPN topology, protection on network traffic is applied and removed by VPN gateways, which act as routers for the two private networks seeking to establish a secure connection. This approach is ideal for organizations that need to connect distant sites in a transparent and secure manner. The hosts within the protected networks are unaware of the existence of the VPN channel, allowing seamless communication between the sites.
- Host-to-Network: In this VPN topology, the VPN tunnel is established between the user's endpoint and the VPN server, granting the user access to

remote private networks. This setup is typically employed for remote workers, enabling secure access to data and applications hosted in corporate data centers and headquarters.

2.2.4 VPN Tunneling Protocols

When considering the landscape of VPN technologies for tunnel creation, there is a wide range of options available, encompassing both standardized and non-standardized solutions. In this thesis, the two primary technologies employed for this task this will be briefly explored: IPsec and TLS.

IPsec

Internet Protocol Security (IPsec) is a widely used IETF architecture that provides OSI Layer 3 security for both IPv4 and IPv6, commonly used for establishing secure VPNs over untrusted networks and ensuring end-to-end secure packet flows. IPsec offers numerous valuable and desirable security properties, including data integrity, sender authentication, partial protection against replay attacks, and confidentiality through encryption. These features make IPsec one of the most reliable, robust, and consequently, popular technologies for secure communications.

IPsec's functionality is based on the concept of Security Associations (SAs) and Security Policies (SPs).
A Security Association (SA) is a unidirectional logical connection between two IPsec systems involved in the communication. It defines the protection mechanisms and keys to be used for the subsequent data transfer. To fully protect a bidirectional packet flow, two SAs are required—one for each direction of communication. In particular, a SA provides information about the following parameters:

- IP addresses of the peers involved in the communication;
- Protocol used for tunnel creation (AH or ESP);
- Cryptographic algorithms and associated keys;
- A 32-bit integer known as the Security Parameter Index (SPI);

A Security Policy (SP) is a rule that determines which traffic must be encapsulated and routed through the tunnel, meaning the traffic that must be protected using IPsec with the information provided by the SA. The SP contains information about:

- IP addresses of the peers involved in the communication;
- Protocol of the network traffic and the corresponding port;
- Identifier of the SA to be used for processing the data;

Both SAs and SPs of a IPsec system are stored in its local databases, respectively Security Association Database (SAD) and Security Policy Database (SPD).

IPsec can be viewed as a collection of multiple protocols, each serving distinct functions:

- Authentication Header (AH) ensures data integrity, sender authentication, protection against IP header modification, and defense against replay attacks;
- Encapsulating Security Payload (ESP) provides data confidentiality, as well as authentication, though it is not equivalent to the level of authentication provided by the AH;
- Internet Security Association and Key Management Protocol (ISAKMP) establishes a framework for authentication and key exchange, with the primary goal of generating the SA;

The IPsec protocols AH and ESP can be implemented in two modes of operation: transport mode and tunnel mode.
Transport mode is used for end-to-end security (so no gateways are usually involved) and only the payload of the IP packet is encrypted or authenticated. It is computationally light, but, since the header is neither modified or encrypted, no protection of header variable fields is provided. The following figures show how the transport mode is applied on AH and ESP protocols:

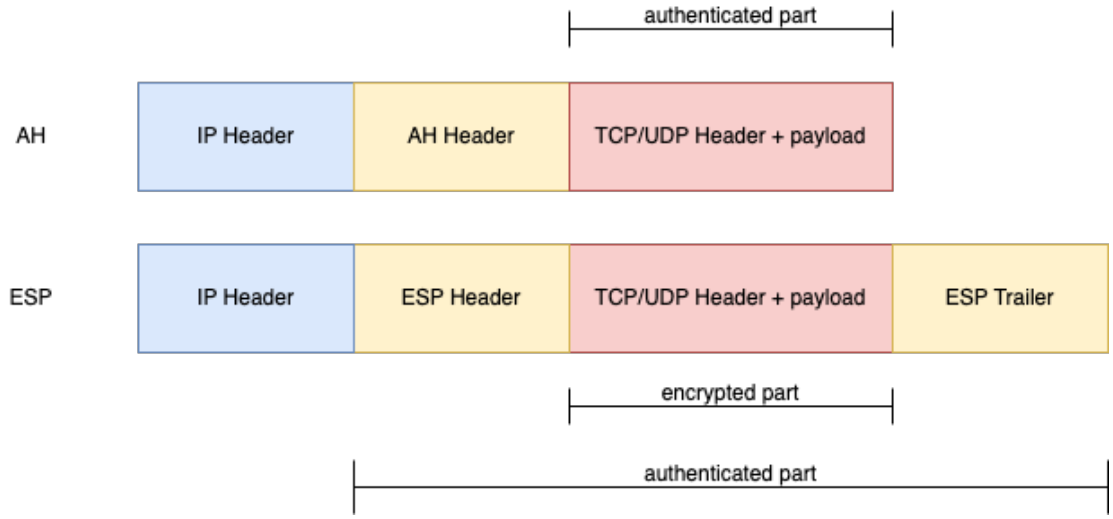


Figure 2.2: Transport mode applied on AH and ESP

On the other hand, with tunnel mode the entire IP packet is encrypted and authenticated. It is then encapsulated into a new IP packet, which includes a new IP header, allowing for the creation of tunnels for VPNs. The following figures show how the tunnel mode is applied on AH and ESP protocols:

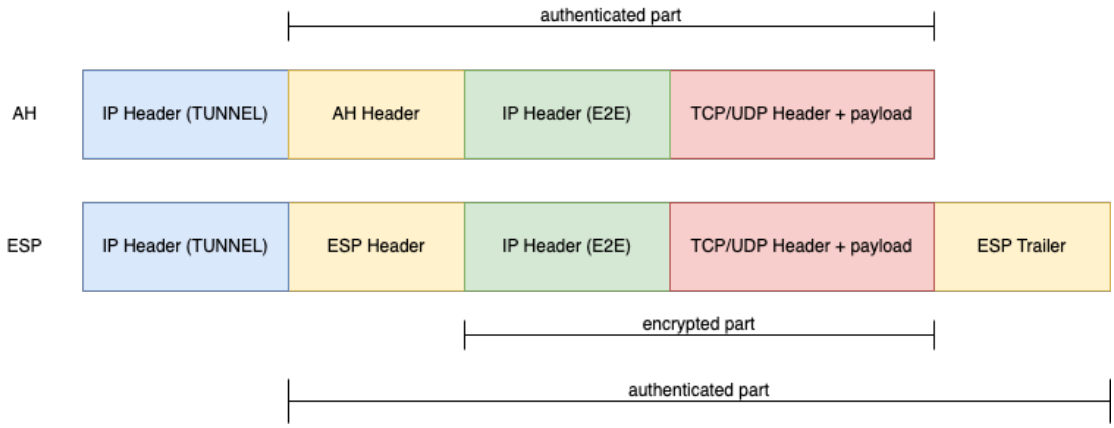


Figure 2.3: Tunnel mode applied on AH and ESP

Finally, one of the key strengths of IPsec is its support for a comprehensive and robust ciphersuite, which ensures a high level of protection for the traffic it secures. It guarantees data integrity and authenticity with HMAC-SHA1/SHA2, confidentiality via TripleDES-CBC and AES-CBC, and authentication through RSA, ECDSA, and EdDSA. Key exchange is handled by Diffie-Hellman and ECDH algorithms.

TLS

Transport Layer Security (TLS) is a cryptographic protocol designed to provide secure communication over a network. It ensures peer authentication through an asymmetric challenge-response mechanism, message confidentiality via symmetric encryption, message authentication and integrity through the computation of a MAC, and protection against replay and filtering attacks using an implicit sequence number, furtherly improving this capability inherited by layering on top of TCP. These properties have established TLS as one of the most widely used protocols for securing various applications, including email services, messaging platforms, VoIP communications, financial transactions, remote desktop protocols, and file transfer mechanisms. However, its role in securing HTTPS remains its most prominent and publicly recognized application.

In recent years, one of the most significant advancement in securing private communication over untrusted networks has been the integration of TLS in the creation of VPNs. A prominent implementation of TLS-based VPNs is OpenVPN, a widely used open-source program that leverages the TLS protocol for cryptographic operations. What TLS brings to the table is primarily its handshake protocol for key exchange and encryption, which enhances the security of VPN connections. TLS initiates a handshake between the client and server, during which they authenticate each other, agree on encryption algorithms, and securely exchange cryptographic keys. Once the handshake is complete, all subsequent data is encrypted using the agreed-upon keys, ensuring secure communication between the VPN client and server. Additionally, TLS supports certificate-based authentication, adding an extra layer of security by verifying the identities of the entities involved in the communication.

The most evident advantages of adopting TLS in the VPN context are its flexibility and compatibility. Unlike IPsec-based VPNs, which operate at the network layer, TLS-based VPNs function at the transport layer, making them more versatile. This allows them to easily traverse firewalls and NAT configurations, simplifying deployment.
The enhanced compatibility is due to the fact that TLS is widely used in securing web traffic (e.g., HTTPS), meaning most devices and networks already support it. As a result, it is easier to establish VPN connections without specialized configurations.

2.3 VEREFOO Overview

As highlighted by the literature review in subsection 2.1.2, VEREFOO is the first automated methodology that integrates formal correctness and verification with optimality in the allocation and configuration of Network Security Functions across both traditional and virtual networks. Specifically, in the realm of VPN configuration, it stands as the first approach to tackle the problem of automatically determining both the allocation scheme and the protection rules for Communication Protection Systems within a network topology, starting from a set of communication protection policies.

2.3.1 General Inputs and Outputs of the Framework

The framework requires the user to provide two inputs: a Service Graph (SG) and the set of Network Security Requirements (NSR).

A Service Graph is a logical representation of the relationships and connections between different network services or functions within a network architecture. The provided SG is automatically processed by the framework and an Allocation Graph is generated, which is the original SG enriched with special nodes called Allocation Places (AP). These special nodes are generated for each link between every pair of edges of the SG and represent the placeholder positions where the needed NSFs (CPSs in the case of VPNs configuration) may be allocated in order to enforce the user-specified NSRs.

The second input is the set of Network Security Requirements that the user desire to be enforced on the SG. In the VPN context, these are referred to as Communication Protection Policies (CPPs), which represent the security requirements for communication protection. They are expressed in a medium-level language to ensure sufficient user-friendliness and to remain independent of specific implementations. Each CPP includes details about policy conditions (IP addresses and ports of the network traffic's source and destination), the algorithms required for ensuring confidentiality and integrity, the VPN protocols and enforcement modes to be used, and, finally, any trustworthiness and inspection requirements for specific nodes and links.

Once the required inputs are provided, the framework moves forward to solve the automatic security enforcement problem. If a solution is found, the framework

outputs the CPS allocation scheme, detailing which APs and other nodes should host the CPSs, along with the protection rules for each allocated CPS. Otherwhise, if no solution is available, a non-enforceability report is generated, providing useful information to help the user understand why the CPPs could not be enforced.

2.3.2 MaxSMT Problem Formulation

The core of this methodology involves formulating the auto-configuration problem as a Maximum Satisfiability Modulo Theories (MaxSMT) problem, which ensures both formal correctness by construction and optimization of the results. Unlike a traditional SAT problem, the MaxSMT approach allows for the inclusion of soft constraints alongside hard constraints. Hard constraints are non-relaxable and must all be satisfied to obtain a solution. Soft constraints, on the contrary, are not strictly required for a valid output; instead, they are assigned weights, with the goal being to maximize the total sum of these weights. In other terms, satisfying the hard constraints ensures formal correctness, while the inclusion of soft constraints enables the framework to achieve optimization goals.

For the automatic configuration of VPNs, hard constraints are used to enforce CPPs and specify the conditions under which a CPS must be allocated on a particular node. Soft constraints, on the other hand, are defined to achieve an optimal solution by minimizing the number of allocated CPSs and the number of configured protection rules. The actual algorithms responsible for the generation of such constraints will be deeply analyzed in the subsequent chapters of the thesis.

For solving the MaxSMT problem, the chosen solver is the *z3* SMT solver developed by Microsoft [33]. This solver was selected due to its efficiency, versatility, and strong support for various theories, in addition to being open-source and freely available.

2.3.3 Communication Protection Systems Model

CPSs are characterized by a set of rules that determine the actions they must perform on specific traffic classes. To enable the SMT solver to compute the necessary rules for enforcing the CPPs, these rules are represented as a set of free variables, for which the solver will determine the actual values, in line with the defined soft and hard constraints. This set of rules associated with each CPS is referred to as the set of *placeholder rules*, as they are generated during the framework's execution, and it is not known in advance which rules the solver will ultimately configure for the final output.

The rules specify the conditions (IP addresses, ports, and protocol) that identify the traffic each rule applies to, the algorithms for enforcing confidentiality and integrity, the VPN protocol to be used (either IPsec or TLS), how the CPS should enforce protection properties, whether the traffic should be encapsulated in a VPN tunnel, and whether the CPS handling the traffic should apply or remove protection.

When the solver allocates a CPS on the graph, a VPN Gateway is placed on that node, and the associated rules are configured and translated in the output

as Security Associations. According to the configured rules, the VPN Gateway is assigned an *ACCESS* behavior, meaning that the node acts as the entry point for the secure channel, or *EXIT* behavior, if the VPN Gateway represents the end of the secure channel. The Security Associations specify the conditions to be matched by the ingress packet class and the actions to be applied on it by the VPN Gateway if the conditions are satisfied.

Chapter 3

Thesis Objective

As mentioned in the introduction, in the realm of network security even minor configuration errors can lead to severe security vulnerabilities, often resulting in breaches and financial losses. Automation offers a way to minimize human error and optimize network configurations.

One tool that embodies this philosophy is VEREFOO, a framework capable of automatically allocating and configuring the necessary security functions on a network to enforce a set of desired security requirements. In this way, the user only needs to define the security policies—referred to as Communication Protection Policies (CPP) in the VEREFOO environment—by specifying the source and destination of the communication to be protected, along with the type of requirement. Depending on the requirement type, different network security functions are allocated and configured automatically by the framework, while also optimizing for performance in terms of solution computation, network bandwidth, and resource allocation.

The framework has achieved significant success in the literature regarding the automated configuration of packet filters, and another version focused on VPNs has been developed in response to the growing importance of this technology for securing communication within organizational environments. However, the current implementation of this VPN-focused version of VEREFOO has not yet matched the results and performance of the version dedicated to packet filters. This is because the task of automatically configuring VPNs is significantly more complex than configuring firewalls. In particular, the formalization of traffic tunneling and VPN behaviors involves numerous factors to be constrained that considerably slow down the computation of the solution.

This thesis aims to investigate the low-level details of the framework's implementation, understand the challenges and issues affecting it, and develop solutions to address them. Specifically, the objectives of this thesis will be pursued through the following steps:

- Presenting a high-level overview of the framework's functioning, with a focus on the key modules that contribute to solution computation and their roles within the VEREFOO ecosystem.

- Conducting an in-depth analysis of each algorithm responsible for constraint generation, which forms the core of the framework's functioning, addressing both network function-related constraints and protection requirement-related constraints.
- Identifying and categorizing issues within the current implementation of the framework, including an assessment of their impact on the correctness and quality of the output.
- Developing solutions for the identified issues and proposing new optimization strategies to enhance VEREFOO's performance.
- Evaluating the performance and testing the effectiveness of the solutions through a specifically designed battery of test cases.

By following these steps, the aim is to enhance VEREFOO's implementation for VPN configuration, contributing to cybersecurity automation by making the tool more reliable, performant, and ultimately a more practical solution to the problem of automatically securing network communications.

Chapter 4

Analysis of VPNs Automatic Configuration Capability

This chapter presents an analysis of the current implementation of the VEREFOO framework's capability to configure secure communication through VPNs. The analysis begins with a high-level overview of the framework's functionality, highlighting its key components and their roles within the VEREFOO ecosystem.

Next, the focus shifts to the core of the framework: the generation of constraints. The source code of each algorithm involved in this process will be examined in detail.

Finally, the chapter will explain the Z3 model produced from these constraints and the framework's final output, with examples provided for illustration.

4.1 High-Level View of Framework Functioning

4.1.1 XML Input File schema

The VEREFOO framework takes as input an XML file that describes the Service Graph (SG), representing the network topology that the framework is responsible for managing, along with a list of Network Security Requirements (NSR) that must be enforced.
The following snippet of code will clearly show the structure of the input file:

```
1   <NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
2       <graphs>
3           <graph id="0">
4           <node functional_type="" name="">
5               <neighbour name ="" />
6               ...
7           </node>
8           ...
9           </graph>
```

```
10        </graphs>
11        <PropertyDefinition>
12            <Property graph="0" name="" src="" dst="">
13                <protectionInfo>
14                    ...
15                <protectionInfo>
16            </Property>
17            ...
18        </PropertyDefinition>
19    <\NFV>
```

Listing 4.1: XML input file structure

In this template, several key definitions can be distinguished:

- `<graphs>`: This element defines multiple network topologies, each identified by an integer number.
- `<node>`: Each node is characterized by two fields. The **functional_type** field defines the type of node, which can be set to *WEBCLIENT*, *WEBSERVER*, *FORWARDER* and many more, or it can be omitted if the node has no predefined role (in which case, it may be considered as a candidate for a *VPN_Gateway* automatic placement). The **name** field specifies the IP address of the node.
- `<neighbour>`: This element is defined within a node and specifies the IP address of each neighbor connected to that node.
- `<Property>`: This element has several fields. The **graph** field specifies the identifier of the graph on which the security property must be enforced. The **name** field accepts values such as *ProtectionProperty*, *IsolationProperty*, and *ReachabilityProperty*, defining the type of security property. The **src** and **dst** fields indicate the IP addresses of the nodes affected by the property, while **src_port** and **dst_port** the number or interval for the ports, and finally **l4_proto** the type of layer 4 protocol (*TCP*, *UDP*, *ANY* or *OTHER*).
- `<protectionInfo>`: This element is defined within the `<Property>` element and contains additional information about the requested protection property.
  - `<authenticationAlgorithm>`: Specifies the authentication algorithm to be used.
  - `<encryptionAlgorithm>`: Specifies the encryption algorithm to be applied.
  - `<inspectorNode>`: Contains the list of inspector nodes, where traffic should not be protected when these nodes are traversed.
  - `<untrustedNode>`: Contains the list of untrusted nodes, requiring traffic protection when these nodes are traversed.
  - `<untrustedLink>`: Similar to untrusted nodes, but applies to links that require protection when traversed.

- `<securityTechnology>`: Specifies the security technology to be used (e.g., IPsec or TLS).
- `<securityTechnologyInfo>`: Contains additional details about the selected security technology, specifying the **mode** (e.g., *Tunnel*, *Transport*, or *Null*) and the **architecture** (e.g., *ESP*, *AH*, etc.).
- `<requiredProtection>`: Specifies the type of protection that must be applied to the traffic (e.g., *Confidentiality Internal*, *Header Integrity Internal Header*, etc.).

4.1.2 Key Modules Overview

In a single iteration of the automatic configuration and allocation of VPNs through the VEREFOO framework, many modules are being called. After receiving in input the XML file containing the SG and list of NSRs, the functioning of the framework can be schematized with the following simplified Module Dependency Graph:

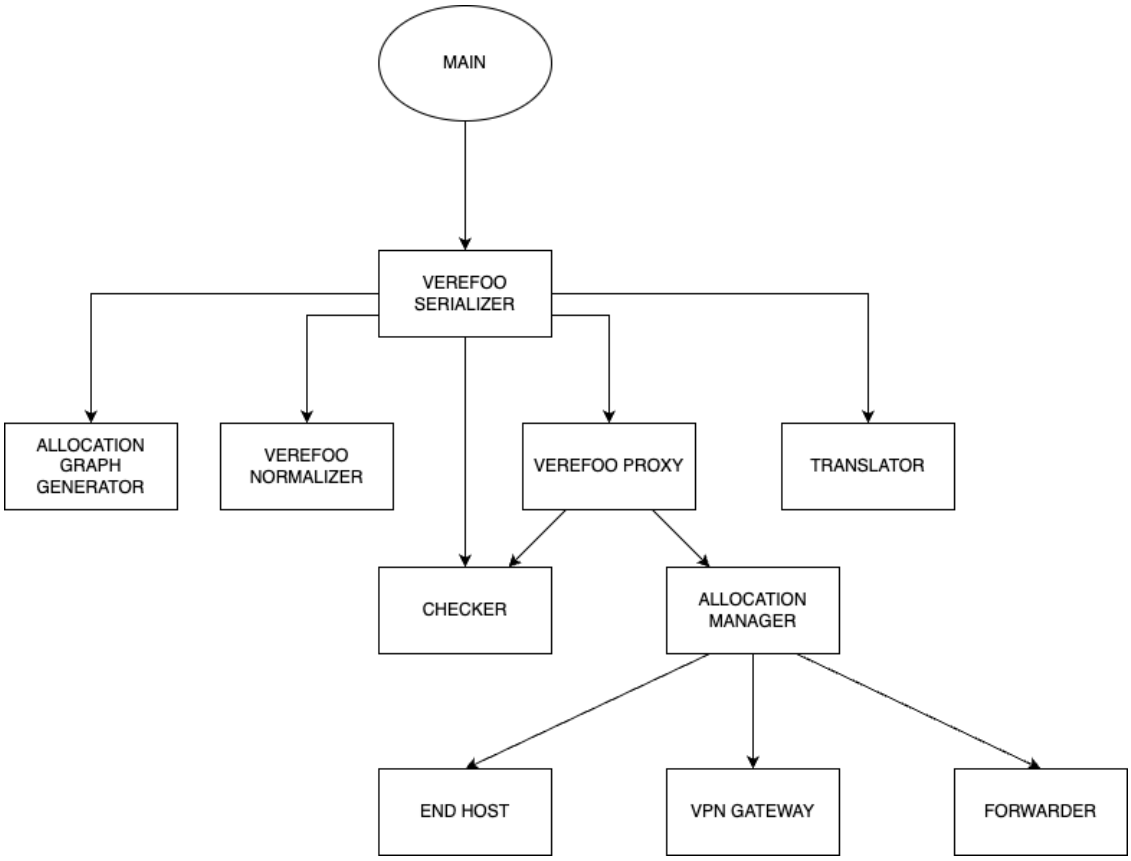


Figure 4.1: Simplified Module Dependency Graph

The figure above highlights the key modules utilized by the framework and their dependency relationships, although many additional modules with smaller roles also

contribute to streamline and enhance the automatic allocation and configuration process.

- **Main**: Represents the entry point of the framework, used to facilitate the debugging process.
- **Verefoo Serializer**: This module wraps all the framework tasks, executing the constraint-solving procedure for each graph in the NFV element of the input file.
- **Verefoo Normalizer**: This module normalizes the security properties, handling any anomalies introduced by the user during their definition and translating them in a more functional format for the tool.
- **Allocation Graph Generator**: Called by the Verefoo Serializer, this module automatically generates the Allocation Graph, starting from the Service Graph (SG) specified by the user as input.
- **Verefoo Proxy**: Considered the most important module, it acts as the interface with other modules and is responsible for:
  - Computing all possible paths related to each requirement.
  - Requesting the instantiation and allocation of functions.
  - Requesting the definition of constraints for each function and requirement.
- **Allocation Manager**: Called by the Verefoo Proxy, this module manages the allocation and deployment of Network Functions on the nodes of the Allocation Graph and requests their configuration by calling the dedicated function configuration modules.
- **Function Configuration Module**: These modules configure and define the soft and hard constraints for each Network Security Function deployed by the Allocation Manager. Each module is specific to a function (e.g., *VPNGateway*, *EndHost*, *Forwarder*, etc.).
- **Checker**: This module defines the hard constraints necessary to enforce the security properties requested by the user. It then submits the complete set of constraints, including those related to function configurations, to the MaxSMT solver to generate a model that satisfies the specified requirements.
- **Translator**: Called by the Verefoo Serializer after the Checker has provided the solution to the MaxSMT problem, this module implements a parser for the output model, translating it into the correct XML file.

4.2 Algorithms for Constraint Generation

Unquestionably, the core of the framework lies in formulating the allocation and configuration problem as a MaxSMT problem, which is achieved through the generation of constraints. As thoroughly explained in section 2.3, the constraints can be classified into two categories:

- **Hard**: These must all be satisfied; otherwise, the MaxSMT problem is declared *UNSATISFIABLE*.
- **Soft**: While not mandatory, the solver aims to satisfy as many of these as possible to maximize the total weight assigned to them, allowing the achievement of an optimized configuration.

It is evident that, in the case of VPN configuration, the constraints dictate where (and whether) the CPS are allocated, how they are configured, and how the traffic must be handled and tunneled to securely reach its destination.

In the framework's source code, the generation of constraints is concentrated into a few key components:

1. **Function Modules**: Responsible for generating constraints related to function allocation, configuration, and VPN tunnel establishment.
2. **Checker Module**: Generates constraints concerning the application or removal of protection on network traffic, depending on the functional typology and trustworthyness of the nodes it traverses.

4.2.1 Network Functions Related Constraints

Since this thesis focuses on the automatic configuration and allocation of VPNs, VEREFOO's capability to handle other Network Functions, such as NATs, Packet Filters, and Loadbalancers, will be disregarded. The code analysis and subsequent enhancements will concentrate on the three fundamental functions that comprise a basic VPN architecture: *VPN Gateway*, *Forwarder*, and *EndHost*.

VPN Gateway

The VPN Gateway is a node specialized in creating VPN tunnels and applying or removing security protection on the traffic it handles. In the VEREFOO context, a VPN Gateway can either be pre-existing in the Service Graph provided by the user or automatically allocated by the framework on the nodes deemed most suitable for hosting one. This type of node is one of only two on which a CPS can be allocated by the framework (the other being the EndHost, if it supports VPN technologies). As a result, the majority of the constraints related to the configuration and allocation of CPS are focused on this module. These constraints can be classified in three main categrories, and will be examined in this order:

1. Placeholder Rule Constraints
2. Allocation Constraints
3. Protection Requirement Satisfiability Constraints

Each allocated CPS is defined by a set of rules that dictate the specific actions it must perform on different packet classes. These rules are represented by a series of free variables, for which the MaxSMT solver is responsible for determining actual values. More specifically, a set of these variables is associated to each node where a CPS may be allocated, and the rules associated with these sets are referred to as "placeholder rules." This terminology is used because, at the time the problem is formulated, it is uncertain whether these rules will ultimately be applied.
The following code snippet illustrates how the placeholder rules are defined through Z3 variables:

```
1     Expr src = ctx.mkConst(vpnGw + "_auto_src_"+flow.getIdFlow(),
          nctx.addressType);
2     Expr dst = ctx.mkConst(vpnGw + "_auto_dst_"+flow.getIdFlow(),
          nctx.addressType);
3     Expr proto = ctx.mkConst(vpnGw +
          "_auto_proto_"+flow.getIdFlow(), ctx.mkIntSort());
4     Expr srcp = ctx.mkConst(vpnGw + "_auto_srcp_"+flow.getIdFlow(),
          nctx.portType);
5     Expr dstp = ctx.mkConst(vpnGw + "_auto_dstp_"+flow.getIdFlow(),
          nctx.portType);
6     Expr authAlg = ctx.mkConst(vpnGw +
          "_auto_auth_alg_"+flow.getIdFlow(), ctx.mkStringSort());
7     Expr encAlg = ctx.mkConst(vpnGw +
          "_auto_enc_alg_"+flow.getIdFlow(), ctx.mkStringSort());
8
9     IntExpr srcAuto1 = ctx.mkIntConst(vpnGw +
          "_auto_src_ip_1_"+flow.getIdFlow());
10    IntExpr srcAuto2 = ctx.mkIntConst(vpnGw +
          "_auto_src_ip_2_"+flow.getIdFlow());
11    IntExpr srcAuto3 = ctx.mkIntConst(vpnGw +
          "_auto_src_ip_3_"+flow.getIdFlow());
12    IntExpr srcAuto4 = ctx.mkIntConst(vpnGw +
          "_auto_src_ip_4_"+flow.getIdFlow());
13    IntExpr dstAuto1 = ctx.mkIntConst(vpnGw +
          "_auto_dst_ip_1_"+flow.getIdFlow());
14    IntExpr dstAuto2 = ctx.mkIntConst(vpnGw +
          "_auto_dst_ip_2_"+flow.getIdFlow());
15    IntExpr dstAuto3 = ctx.mkIntConst(vpnGw +
          "_auto_dst_ip_3_"+flow.getIdFlow());
16    IntExpr dstAuto4 = ctx.mkIntConst(vpnGw +
          "_auto_dst_ip_4_"+flow.getIdFlow());
17
18    ...
19
20    srcConditions.put(i, src);
```

```
21      dstConditions.put(i, dst);
22      portSConditions.put(i, srcp);
23      portDConditions.put(i, dstp);
24      l4Conditions.put(i, proto) ;
25      authAlgConditions.put(i, authAlg);
26      encAlgConditions.put(i, encAlg);
```

Listing 4.2: Placeholder rule definition through Z3 variables

Generally, the assertions are structured as follows: a string representing the type of node (e.g., *vpnGw*), a string representing the variable (e.g., *_auto_src_*, *_auto_dst_*, etc.), the identifier of the flow, and finally, the type of variable (e.g., *addressType*, *portType*, etc.). In the second part of the reported code, the source and destination addresses are split into the four parts that make up the complete address. This approach facilitates easier address management within the framework. Finally, the variables are inserted into their respective HashMaps, where the key is an integer that increments for each placeholder rule.

After fully defining the placeholder rules through the set of Z3 variables, the framework proceeds to generate the following constraints on these rules:

```
1      implications1.add(ctx.mkAnd(ctx.mkNot(ctx.mkEq( src,
          this.nctx.addressMap.get("null"))),
2      ctx.mkNot(ctx.mkEq( dst, this.nctx.addressMap.get("null")))));
3      implications2.add(ctx.mkAnd(ctx.mkEq( src,
          this.nctx.addressMap.get("null")),
4      ctx.mkEq( dst, this.nctx.addressMap.get("null"))));
```

Listing 4.3: Constraints on placeholder rules

The purpose of these constraints is to ensure that the destination of the rule is set to *null* if the source is *null*, and to ensure that the destination is **not** *null* if the source is also not *null*.

For the allocation of a CPS on the VPN Gateway, the following constraints are designed to minimize their allocation. This approach helps achieve the optimization goal of maximizing the available network bandwidth while ensuring that VPN tunnels are present to protect the traffic:

```
1      nctx.softConstrVPNGWAutoPlace.add(new Tuple<BoolExpr,
          String>(ctx.mkNot(used), "VPN_auto_conf"));
2      ...
3      List<BoolExpr> uses = new ArrayList<>();
4      for(Map.Entry<Integer,BoolExpr> entry :
          usedVPNFlowsMap.entrySet()) {
5          uses.add(entry.getValue());
6      }
7
8      BoolExpr[] tmp = new BoolExpr[uses.size()];
```

```
9     BoolExpr totalUse = ctx.mkAtLeast(uses.toArray(tmp),1) ;
10
11    constraints.add((BoolExpr)ctx.mkITE(totalUse, ctx.mkEq(used,
          ctx.mkTrue()), ctx.mkEq(used, ctx.mkFalse())) );
```

Listing 4.4: Constraints on CPS allocation on VPN Gateway

- **Line 1**: this soft constraint state that, if possible, the VPN Gateway should not be used. The soft constraint is assigned to the group *VPN_auto_conf* and of the type *softConstrVPNGWAutoPlace*, which determines the weight assigned to the constraint.
- **Line 3-11**: this hard constraint states that if the node has at least one communication protection rule configured, it implies that a VPN Gateway **must** be allocated on that node.

Finally, the third category of constraints for the VPN Gateway is managed by the `generateSatisfiabilityConstraint()` function, which is called for every traffic flow crossing the node. The first step of the function is to generate constraints about the VPN Gateway behavior:

```
1     BoolExpr opt1 = ctx.mkAnd(
          ctx.mkEq(nctx.protect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkTrue()),
          ctx.mkEq(nctx.disProtect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()) );
2     BoolExpr opt2 = ctx.mkAnd(
          ctx.mkEq(nctx.protect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()),
          ctx.mkEq(nctx.disProtect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkTrue()) );
3     BoolExpr opt3 = ctx.mkAnd(
          ctx.mkEq(nctx.protect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()),
          ctx.mkEq(nctx.disProtect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()) );
4     BoolExpr behaviour = ctx.mkOr(opt1,opt2,opt3);
5     constraints.add(behaviour);
6     constraints.add((BoolExpr)ctx.mkITE(opt3, ctx.mkEq(currentUse,
          ctx.mkFalse()), ctx.mkEq(currentUse, ctx.mkTrue())));
```

Listing 4.5: Constraints on VPN Gateway behavior

- **Lines 1-3**: these lines formally define the three behaviors that can be assigned to the VPN Gateway.

  - **opt1**: the first boolean expression defines the *ACCESS* behavior, which means that the node applies protection on the traffic flow (i.e., *protect=TRUE* and *disProtect=FALSE*);
  - **opt2**: the second boolean expression defines the *EXIT* behavior, which mean that the protection removes protection on the traffic flow (i.e., *protect=FALSE* and *disProtect=TRUE*);
  - **opt3**: the third boolean expression defines the *FORWARD* behavior, which means that the node does not apply or remove protection on the traffic flow (i.e., *protect=FALSE* and *disProtect=FALSE*);

- **Lines 4-5**: the hard constraint states that one of the three possible behaviors must be assigned to the VPN Gateway.

- **Line 6**: the hard constraint states that if the assigned behavior is *NOT USED*, the value of the free variable *used* is forced to false, otherwhise, either with *ACCESS* or *EXIT*, the value of *used* is forced to true.

In the second part of the `generateSatisfiabilityConstraint()` function, the construction of VPN tunnels is formulated through a set of constraints. However, before analyzing this set of constraints, it is helpful to explain how VEREFOO manages VPN tunnels.

In the **Flow** class of VEREFOO, for each node along the flow path, a pair of Z3 variables representing the tunnel source and tunnel destination addresses are created and stored in two HashMaps: *tunnelSrcMap* and *tunnelDstMap*.

```
1     this.tunnelSrcMap = new HashMap<String, Expr>();
2     this.tunnelDstMap = new HashMap<String, Expr>();
3     this.tunnelMap = new HashMap<String, VPNTunnel>();
4     for(int i=0; i<this.path.getNodes().size(); i++) {
5         Expr tunnelSrc = ctx.mkConst(idFlow + "_auto_tunnelSrc_"+i,
              nctx.addressType);
6         Expr tunnelDst = ctx.mkConst(idFlow + "_auto_tunnelDst_"+i,
              nctx.addressType);
7         String nodeName =
              this.path.getNodes().get(i).getNode().getName();
8         this.tunnelSrcMap.put(nodeName, tunnelSrc);
9         this.tunnelDstMap.put(nodeName, tunnelDst);
10    }
```

Listing 4.6: Tunnel handling through Z3 variable definition

In these maps, the key is the node's address, and the value is the corresponding Z3 variable, which is structured as the concatenation of the flow identifier, the string *_auto_tunnelSrc_* or *_auto_tunnelDst_*, and an integer representing the position of the node in the path. For example, the free variable *"0_auto_tunnelSrc_0"* refers to the tunnel source address of the first node in the flow with identifier *"0"*. This structure allows the framework to establish VPN tunnels by assigning addresses

to the source and destination variables of the tunnels, based on the constraints generated in the VPN Gateway class.

Returning to the topic of constraint generation, the assignment of VPN tunnel source and destination addresses is deeply linked to the behavior of the VPN gateway.

```
1     BoolExpr tunnelSrcStartConstruct =
         nctx.equalIpAddressToVPNGwSA(source.getNode().getName(),
         flow.getTunnelSrc(nextNode)) ;
2     Traffic t = flow.getCrossedTraffic(source.getNode().getName());
3     BoolExpr tunnelSrcEndConstruct =
         nctx.equalIpAddressToVPNGwSA(t.getIPSrc(),
         flow.getTunnelSrc(nextNode)) ;
4     BoolExpr tunnelSrcForwardConstruct =
         ctx.mkEq(flow.getTunnelSrc(source.getNode().getName()),
         flow.getTunnelSrc(nextNode)) ;
5
6     List<AllocationNode> vpnExits = new ArrayList<>();
7     for( int i=currentVPNpos+1; i<flow.getPath().getNodes().size();
         i++ ) {
8         if(flow.getPath().getNodes().get(i).getPlacedNF()
9         .getHasVpnCapability()){
10            vpnExits.add(flow.getPath().getNodes().get(i));
11        }
12    }
13    List<BoolExpr> tunnelDstStartConstructs = new ArrayList<>();
14    for(AllocationNode vpnexit : vpnExits) {
15        BoolExpr opt2VPNExit = ctx.mkAnd(
              ctx.mkEq(nctx.protect.apply(vpnexit.getZ3Name(),
              ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()),
              ctx.mkEq(nctx.disProtect.apply(vpnexit.getZ3Name(),
              ctx.mkInt(flow.getIdFlow())), ctx.mkTrue()) );
16        tunnelDstStartConstructs.add(ctx.mkAnd(opt2VPNExit,
              nctx.equalIpAddressToVPNGwSA(vpnexit.getNode().getName(),
              flow.getTunnelDst(nextNode)),
              ctx.mkEq(flow.getTunnelSrc(vpnexit.getNode().getName()),
              flow.getTunnelSrc(nextNode)) ) );
17    }
18    BoolExpr[] temp = new BoolExpr[tunnelDstStartConstructs.size()];
19    BoolExpr tunnelDstStartConstruct =
         ctx.mkOr(tunnelDstStartConstructs.toArray(temp)) ;
20    BoolExpr tunnelDstEndConstruct =
         nctx.equalIpAddressToVPNGwSA(t.getIPDst(),
         flow.getTunnelDst(nextNode)) ;
21    BoolExpr tunnelDstForwardConstruct =
         ctx.mkEq(flow.getTunnelDst(source.getNode().getName()),
         flow.getTunnelDst(nextNode)) ;
```

```
22    BoolExpr tunnelSrcStartConfig = ctx.mkImplies(opt1,
          tunnelSrcStartConstruct);
23    constraints.add(tunnelSrcStartConfig);
24    BoolExpr tunnelDstStartConfig = ctx.mkImplies(opt1,
          tunnelDstStartConstruct);
25    constraints.add(tunnelDstStartConfig);
26    BoolExpr tunnelSrcEndConfig = ctx.mkImplies(opt2,
          tunnelSrcEndConstruct);
27    constraints.add(tunnelSrcEndConfig);
28    BoolExpr tunnelDstEndConfig = ctx.mkImplies(opt2,
          tunnelDstEndConstruct);
29    constraints.add(tunnelDstEndConfig);
30    BoolExpr tunnelSrcForwardConfig = ctx.mkImplies(opt3,
          tunnelSrcForwardConstruct);
31    constraints.add(tunnelSrcForwardConfig);
32    BoolExpr tunnelDstForwardConfig = ctx.mkImplies(opt3,
          tunnelDstForwardConstruct);
33    constraints.add(tunnelDstForwardConfig);
```

Listing 4.7: Constraints on source and destination tunnel addresses

- **Source Tunnel Address**
  - *ACCESS* behavior: if the VPN Gateway has been assigned an *ACCESS* behavior, then the tunnel source address of the next node of the flow path must be the address of the VPN Gateway.
  - *EXIT* behavior: if the VPN Gateway has been assigned an *EXIT* behavior, then the tunnel source address of the next node of the flow path must be the source of the traffic itself.
  - *FORWARD* behavior: if the VPN Gateway has been assigned a *FORWARD* behavior, then the tunnel source address of the next node of the flow path must be equal to the tunnel source address assigned to the VPN Gateway.
- **Destination Tunnel Address**
  - *ACCESS* behavior: if the VPN Gateway has been assigned an *ACCESS* behavior, then one of the next VPN Gateway on the flow path must be assigned an *EXIT* behavior, and the tunnel destination address of the next node of the flow path must be equal to the address of the VPN Gateway with the *EXIT* behavior assigned.
  - *EXIT* behavior: if the VPN Gateway has been assigned an *EXIT* behavior, then the tunnel destination address of the next node of the flow path must be the address of the VPN Gateway.
  - *FORWARD* behavior: if the VPN Gateway has been assigned a *FORWARD* behavior, then the tunnel destination address of the next node of the flow path must be equal to the tunnel destination address assigned to the VPN Gateway.

Finally, the function concludes with a final set of constraints related to the Security Association configuration. In this section, if the VPN Gateway is assigned an *ACCESS* or *EXIT* role, the free variables defined in Listing 4.2 are assigned the values corresponding to the actual traffic.

```
1  BoolExpr component2 = ctx.mkAnd(
2  nctx.equalIpAddressToVPNGwSA(flow.getCrossedTraffic(ipAddress)
3      .getIPSrc(), srcConditions.get(srCount)),
4  nctx.equalIpAddressToVPNGwSA(flow.getCrossedTraffic(ipAddress)
5      .getIPDst(), dstConditions.get(srCount)),
6  nctx.equalPort(flow.getCrossedTraffic(ipAddress)
7      .getpSrc(), portSConditions.get(srCount)),
8  nctx.equalPort(flow.getCrossedTraffic(ipAddress)
9      .getpDst(), portDConditions.get(srCount)),
10 nctx.equalLv4Proto(flow.getCrossedTraffic(ipAddress)
11     .gettProto().ordinal(), l4Conditions.get(srCount)));
12 srCount++;
13 constraints.add(ctx.mkImplies(ctx.mkOr(opt1,opt2),component2));
```

Listing 4.8: Constraints on SA configuration

Wrapping everything together, the constraints related to the VPN Gateway are the following:

1. Constraints on placeholder rules, Listing 4.3
2. Constraints on VPN Gateway allocation, Listing 4.4
3. Constraints on protection requirements satisfiability
    - Constraints on VPN Gateway behavior, Listing 4.5
    - Constraints on VPN tunnels, Listing 4.7
    - Constraints on SA configuration, Listing 4.8

Forwarder

The functional type of *Forwarder* is one of the fundamental components of any computer network. In general, a Forwarder is a node that possesses only forwarding behavior rather than transformation behavior, meaning it can determine whether a packet class belonging to a traffic flow is discarded or forwarded to the next hop, without modifying the packet in any way.

Within the VEREFOO framework, the Forwarder functional type is further simplified, as it only forwards packets to the next hops. Other Network Functions, such as Packet Filters and Loadbalancers, are specifically modeled within the framework to handle filtering actions and other advanced traffic management tasks.

The simplicity of this functional type also impacts the constraints required to formalize its behavior, which are limited to ensuring proper forwarding and maintaining consistency for VPN tunnels between adjacent nodes. Specifically, the constraints about its behavior are the following:

```
1    constraints.add(ctx.mkEq(nctx.protect.apply(source.getZ3Name(),
        ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()));
2    constraints.add(ctx.mkEq(nctx.disProtect.apply(source.getZ3Name(),
        ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()));
3    constraints.add(ctx.mkEq(nctx.deny.apply(source.getZ3Name(),
        ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()));
```

Listing 4.9: Constraints on Forwarder behavior

The first two constraints specify that a Forwarder node cannot apply or remove protection on any traffic it handles (i.e., *protect=FALSE* and *disProtect=FALSE*). The third constraint reinforces its forwarding behavior by disabling the discard action (i.e., *deny=FALSE*).

As mentioned earlier, the second set of constraints is related to the VPN tunnels, ensuring that the tunnel addresses remain unchanged and that consistency is maintained throughout the traffic flow path:

```
1    constraints.add(ctx.mkEq(flow.getTunnelSrc(
2        source.getNode().getName()), flow.getTunnelSrc(nextNode)));
3    constraints.add(ctx.mkEq(flow.getTunnelDst(
4        source.getNode().getName()), flow.getTunnelDst(nextNode)));
```

Listing 4.10: Forwarder constraints on tunnels

In particular, these hard constraints set the tunnel source and destination addresses of the next node to the same values assigned to the Forwarder node.

End Host

End Hosts function as the endpoints in a network connection, which can either be a client or a server. In VEREFOO, there is minimal distinction between how clients and servers are implemented. The main difference lies in the FunctionalType assigned when the node is created: clients are designated as *WEBCLIENT*, and servers as *WEBSERVER*. Additionally, both types of nodes have a *vpnCapabilities* attribute. When this attribute is set to *true*, it indicates that the node is capable of acting as a VPN gateway, allowing it to serve as either the starting or ending point of the VPN, thus enabling secure data transmission.

In the context of constraints generation, the *vpnCapabilities* attribute is crucial, as different sets of constraints are generated depending on the VPN capability of the End Host.

Starting from the simpler case, if the EndHost does not support VPN technologies, the following constraints are generated:

```
1    constraints.add(ctx.mkEq(nctx.protect.apply(source.getZ3Name(),
        ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()));
```

```
2     constraints.add(ctx.mkEq(nctx.disProtect.apply(source.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()));
3     constraints.add(ctx.mkEq(vpnCapIsUsed, ctx.mkFalse()));
4
5     if(lastNode==false){
6         Expr tunnelSrc =
              flow.getTunnelSrc(source.getNode().getName());
7         constraints.add(nctx.equalIpAddressToVPNGwSA(t.getIPSrc(),
              tunnelSrc) );
8         constraints.add(ctx.mkEq(flow.getTunnelSrc(source.getIpAddress()),
              flow.getTunnelSrc(nextNode)) );
9         Expr tunnelDst =
              flow.getTunnelDst(source.getNode().getName());
10        Traffic t2 =
              flow.getCrossedTraffic(source.getNode().getName());
11        constraints.add(nctx.equalIpAddressToVPNGwSA(t2.getIPDst(),
              tunnelDst) );
12        constraints.add(ctx.mkEq(flow.getTunnelDst(source.getIpAddress()),
              flow.getTunnelDst(nextNode)) );
13    }
```

Listing 4.11: EndHost constraints with no vpnCapabilities

- **Lines 1-3**: These hard constraints indicate that since the node does not possess VPN capabilities, it cannot apply or remove protection on the traffic it handles, and the Z3 variable *vpnCapabilities* must be set to *false*.
- **Lines 6-13**: These hard constraints apply only if the EndHost is not the last node in the traffic path, meaning it is of the *WEBCLIENT* type and the source of the traffic.
  - **Lines 6-8**: These hard constraints specify that the tunnel source address of the first node of the path must be set to the IP address of the actual traffic's source (which is the node itself). Additionally, the tunnel source address of the next node is forced to be equal to the EndHost's tunnel source address.
  - **Lines 9-12**: These hard constraints specify that the tunnel destination address of the first node of the path must be set to the IP address of the actual traffic's destination. Furthermore, the tunnel destination address of the next node is forced to be equal to the EndHost's tunnel destination address.

In summary, the behavior of the EndHost is described by the first three constraints, while the remaining constraints set the source and destination of the tunnel to match the source and destination of the traffic, and impose consistency between the first node of the path and the next one.

On the other hand, if the EndHost does support VPN technologies, the configuration of the node is much more complex.

The first step is the same reported in Listing 4.2, which is the definition of the free variables describing the fields of the placeholder rules. The only difference is the string used to represent the type of node; in this case, *politoEndHost* is used instead of *vpnGw*. However, the set of constraints applied to the placeholder rules is extended. In addition to the constraints discussed in Listing 4.3, new soft and hard constraints are introduced to further minimize both the allocation of CPS on the network and the number of rules needed to enforce the required protection properties.

To enhance consistency and facilitate easier management of IP addresses within the Z3 model, the following hard constraints ensure that each component of the source and destination addresses of the rule is equal to the previously defined Z3 variables:

```
1     constraints.add(ctx.mkAnd(
2         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_1").apply(src),
              srcAuto1),
3         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_2").apply(src),
              srcAuto2),
4         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_3").apply(src),
              srcAuto3),
5         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_4").apply(src),
              srcAuto4),
6         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_1").apply(dst),
              dstAuto1),
7         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_2").apply(dst),
              dstAuto2),
8         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_3").apply(dst),
              dstAuto3),
9         ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_4").apply(dst),
              dstAuto4))
10        );
```

Listing 4.12: Constraints on rule's IP addresses

Next, to enable a single rule to satisfy multiple security properties simultaneously, the following soft constraints are introduced:

```
1 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
2     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_1")
3     .apply(nctx.addressMap.get("wildcard")),srcAuto1)),
          "vpn_auto_conf"));
4 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
5     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_2")
6     .apply(nctx.addressMap.get("wildcard")),srcAuto2)),
          "vpn_auto_conf"));
7 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
8     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_3")
9     .apply(nctx.addressMap.get("wildcard")),srcAuto3)),
          "vpn_auto_conf"));
```

```
10 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
11     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_4")
12     .apply(nctx.addressMap.get("wildcard")),srcAuto4)),
          "vpn_auto_conf"));
13 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
14     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_1")
15     .apply(nctx.addressMap.get("wildcard")),dstAuto1)),
          "vpn_auto_conf"));
16 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
17     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_2")
18     .apply(nctx.addressMap.get("wildcard")),dstAuto2)),
          "vpn_auto_conf"));
19 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
20     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_3")
21     .apply(nctx.addressMap.get("wildcard")),dstAuto3)),
          "vpn_auto_conf"));
22 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
23     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_4")
24     .apply(nctx.addressMap.get("wildcard")),dstAuto4)),
          "vpn_auto_conf"));
25
26     ...
27
28 nctx.softConstrProtoWildcard.add(new Tuple<BoolExpr, String>(
29     (ctx.mkEq( proto, ctx.mkInt(0)) ),"vpn_auto_conf"));
30 nctx.softConstrPorts.add(new Tuple<BoolExpr, String>(
31     (ctx.mkEq(srcp, nctx.portMap.get("null")) ),"vpn_auto_conf"));
32 nctx.softConstrPorts.add(new Tuple<BoolExpr, String>(
33     (ctx.mkEq(dstp, nctx.portMap.get("null")) ),"vpn_auto_conf"));
```

Listing 4.13: Constraints on wildcard usage

In VEREFOO, the Wildcard can be applied on network addresses, ports and L4 protocols with the following values:

- IP address: -1 on each component of the address;
- Ports: 0-65535;
- Protocol: "ANY";

Finally, to minimize the number of rules that need to be configured, the following soft constraints are generated:

```
1     BoolExpr notConfiguredRule = ctx.mkAnd(
2     ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_1").apply(
3         nctx.addressMap.get("null")), srcAuto1),
4     ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_2").apply(
5         nctx.addressMap.get("null")), srcAuto2),
6     ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_3").apply(
```

```
7         nctx.addressMap.get("null")), srcAuto3),
8     ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_4").apply(
9         nctx.addressMap.get("null")), srcAuto4),
10    ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_1").apply(
11        nctx.addressMap.get("null")), dstAuto1),
12    ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_2").apply(
13        nctx.addressMap.get("null")), dstAuto2),
14    ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_3").apply(
15        nctx.addressMap.get("null")), dstAuto3),
16    ctx.mkEq(nctx.ipFunctionsMap.get("ipAddr_4").apply(
17        nctx.addressMap.get("null")), dstAuto4)
18    );
19    nctx.softConstrAutoConf.add(new Tuple<BoolExpr,
          String>(notConfiguredRule, "vpn_auto_conf"));
20    notConfiguredConditions.put(i, notConfiguredRule);
```

Listing 4.14: Constraints on not configuring rules

By forcing a *NULL* value on both the source and destination addresses simultaneously (when possible), the rule is not generated.

Having covered the constraints related to rule configuration, the next section of constraints addresses the formalization of the EndHost's behavior and tunnel management, depending on its functional type.

If the node is a *WEBCLIENT*, the *ACCESS* and *FORWARD* behaviors, as discussed in Listing 4.5, are defined, while the *EXIT* behavior is disregarded, as the source of the traffic cannot simultaneously serve as the termination point of a VPN tunnel. Regardless of the behavior assigned to the EndHost, the constraints applied to the tunnel source address are identical to those described in lines 6-8 of Listing 4.11, while the tunnel destination address constraints are the same as those outlined in Listing 4.7. The SA configuration is based on the set of constraints analyzed in Listing 4.8, but with a notable difference:

```
1  List<BoolExpr> listSecondC = new ArrayList<>();
2  for(int i = 0; i < nRules; i++) {
3      BoolExpr component2 = ctx.mkAnd(
4              nctx.equalIpAddressToVPNGwSA(
5                  flow.getCrossedTraffic(ipAddress).getIPSrc(),
                       srcConditions.get(i)),
6              nctx.equalIpAddressToVPNGwSA(
7                  flow.getCrossedTraffic(ipAddress).getIPDst(),
                       dstConditions.get(i)),
8              nctx.equalPort(flow.getCrossedTraffic(ipAddress).getpSrc(),
                   portSConditions.get(i)),
9              nctx.equalPort(flow.getCrossedTraffic(ipAddress).getpDst(),
                   portDConditions.get(i)),
10             nctx.equalLv4Proto(flow.getCrossedTraffic(ipAddress)
11                 .gettProto().ordinal(), l4Conditions.get(i)));
12     listSecondC.add(component2);
```

```
13 }
14 BoolExpr[] tmp = new BoolExpr[listSecondC.size()];
15 BoolExpr secondC = ctx.mkOr(listSecondC.toArray(tmp)) ;
16 constraints.add(ctx.mkImplies(opt1,secondC) );
```

Listing 4.15: Constraints on SA configuration by EndHost

In this case, `component2` is defined for each placeholder rule and collected in the list `listSecondC`. All the elements in the list are then combined using a boolean *OR* expression, and a single constraint based on that expression is generated. The constraint is interpreted as: "if the EndHost is assigned an *ACCESS* behavior, then one (or more) SA must be configured." In contrast, in Listing 4.8, a separate constraint is generated for each placeholder rule. The impact of this distinction will be analyzed in a later section of this thesis.

On the other hand, if the EndHost's functional type is *WEBSERVER*, the formalized behaviors are *EXIT* and *NOT USED*. The notation *NOT USED* is preferred in the case of a *WEBSERVER* node, as there are no further hops in the traffic path. However, in the context of VPN capabilities utilization, this behavior is equivalent to the *FORWARD* behavior previously discussed. Since the *WEBSERVER* is the actual destination of the traffic, no constraints on VPN tunnels are introduced. The constraints on the SAs configuration are also the same discussed in Listing 4.15, with the only difference being that the placeholder rules have to be configured if the EndHost is assigned an *EXIT* behavior.

A final set of constraints, (almost) indipendently of the EndHost's functional type, handles the two VPN technologies that can be used to enforce communication protection and establish tunnels: IPsec and TLS.
Starting from the IPsec case:

- If IPsec is supported by the end host (`this.supportedVpnTecnologies` contains "IPSEC"):
  - The framework checks for the different capabilities of IPsec by iterating over several capability types (`IPSECcapability`, `AuthenticationAlgorithmType`, `EncryptionAlgorithmType`).

```
1  for(IPSECcapability ipsc : IPSECcapability.values()) {
2      String string = "IPSEC_" + ipsc;
3      BoolExpr c = ctx.mkEq(nctx.supportCap.apply(
4          source.getZ3Name(),ctx.mkInt(
5              flow.getIdFlow()),ctx.mkString(string)),
                   ctx.mkTrue());
6      ipsecUsedBoolExprs.add(c);
7  }
8  for (AuthenticationAlgorithmType aa :
       AuthenticationAlgorithmType.values()) {
9      if(!ipsecCapList.contains(aa.toString())) {
10         String string = "IPSEC_" + aa.toString();
11         BoolExpr c = ctx.mkEq(nctx.supportCap.apply(
12             source.getZ3Name(),ctx.mkInt(
```

```
13                flow.getIdFlow()),ctx.mkString(string)),
                      ctx.mkFalse());
14            ipsecUsedBoolExprs.add(c);
15        }
16    }
17    for (EncryptionAlgorithmType ea :
          EncryptionAlgorithmType.values()) {
18        if(!ipsecCapList.contains(ea.toString())) {
19            String string = "IPSEC_" + ea.toString();
20            BoolExpr c = ctx.mkEq(nctx.supportCap.apply(
21                source.getZ3Name(),ctx.mkInt(
22                    flow.getIdFlow()),ctx.mkString(string)),
                          ctx.mkFalse());
23            ipsecUsedBoolExprs.add(c);
24        }
25    }
26    for(Capability capability : Capability.values()) {
27        String string = "IPSEC_" + capability;
28        BoolExpr c = ctx.mkEq(nctx.supportCap.apply(
29            source.getZ3Name(),ctx.mkInt(
30                flow.getIdFlow()),ctx.mkString(string) ),
                      ctx.mkFalse());
31        ipsecNotUsedBoolExprs.add(c);
32    }
```

Listing 4.16: Boolean Expression definition of IPsec capabilities of EndHost

  - For each capability, if supported, it generates a Boolean expression indicating the host supports that capability for the traffic flow; otherwhise, a constraint indicating the capability is not supported is added.
  - These expressions are combined into `ipsecUsedFinal` (capabilities used) and `ipsecNotUsedFinal` (capabilities not used), which are combined into the `ipsecFinal` expression using a logical OR.

- Otherwhise, if IPsec is not supported, the code generates constraints that explicitly mark all IPsec-related capabilities as unsupported.

The framework mirrors the discussed management approach for TLS technology, where similar checks are performed for the supported TLS capabilities. Expressions are created to determine whether TLS capabilities are utilized or not, and these are combined into the final expressions (`tlsUsedFinal` and `tlsNotUsedFinal`), which are subsequently merged into `tlsFinal`.

After processing both IPsec and TLS, the code adds the final constraints to the constraints list. This step varies slightly depending on the functional type of the EndHost:

- *WEBCLIENT*

```
1  BoolExpr[] capTmp = new BoolExpr[capConstraints.size()];
2  BoolExpr capConsFinal =
       ctx.mkAnd(capConstraints.toArray(capTmp)) ;
3
4  constraints.add(ctx.mkImplies(opt1,capConsFinal) );
5  if(ipsecFinal != null && tlsFinal != null)
6      constraints.add(ctx.mkOr(ipsecFinal,tlsFinal));
7
8  List<BoolExpr> sameVpnTechConstructs = new ArrayList<>();
9  for(AllocationNode vpnexit : vpnExits) {
10     BoolExpr opt2VPNExit = ctx.mkAnd(
           ctx.mkEq(nctx.protect.apply(vpnexit.getZ3Name(),
           ctx.mkInt(flow.getIdFlow())), ctx.mkFalse()),
11             ctx.mkEq(nctx.disProtect.apply(vpnexit.getZ3Name(),
                   ctx.mkInt(flow.getIdFlow())), ctx.mkTrue())
                   );
12
13     sameVpnTechConstructs.add(ctx.mkAnd(opt2VPNExit,
14             ctx.mkEq(nctx.usedTechnology.apply(
15                 source.getZ3Name(),ctx.mkInt(
16                     flow.getIdFlow()),ctx.mkString("IPSEC")),
                           nctx.usedTechnology.apply(
17                         vpnexit.getZ3Name(),ctx.mkInt(
18                             flow.getIdFlow()),
19                                 ctx.mkString("IPSEC"))),
20             ctx.mkEq(nctx.usedTechnology.apply(
21                 source.getZ3Name(),ctx.mkInt(
22                     flow.getIdFlow()),ctx.mkString("TLS") ),
                           nctx.usedTechnology.apply(
23                         vpnexit.getZ3Name(),ctx.mkInt(
24                             flow.getIdFlow()),
25                                 ctx.mkString("TLS") )),
26             ctx.mkEq(flow.getTunnelSrc(
27                 vpnexit.getNode().getName()),
                       flow.getTunnelSrc(nextNode))));
28 }
29 BoolExpr[] sameVpnTechConstructTemp = new
       BoolExpr[sameVpnTechConstructs.size()];
30 BoolExpr sameVpnTechConstruct = ctx.mkOr(
31     sameVpnTechConstructs.toArray(sameVpnTechConstructTemp));
32 constraints.add(ctx.mkImplies(opt1, sameVpnTechConstruct));
```

Listing 4.17: Constraints on IPsec and TLS capabilities by EndHost

– **Line 1-4**: the constraint ensures that assigning a *ACCESS* behavior to the EndHost implies that all the capability constraints have to be satisfied.

  - **Line 5-6**: the constraint specify that if the node support both the IPsec and TLS, one of the two technologies must be used.
  - **Line 8-32**: This section adds conditions to ensure that the VPN exit behaves correctly in terms of protection, uses the same VPN technology (IPsec or TLS) as the source host, and verifies the correct tunnel source for communication between node.

- *WEBSERVER*
  - In this case, the constraints from lines 1 to 6 remain the same, but the condition is the *EXIT* behavior.
  - Since the node serves as the destination of the traffic, the second section of the code block (lines 8-32) is not required.

To summarize all the constraints related to the EndHost configuration:

- EndHost with no VPN capabilities, Listing 4.11 :
  1. Constraints on behavior
  2. Constraints on tunnels
- EndHost with VPN capabilities:
  1. Constraints on placeholder rules configuration, Listing 4.12 Listing 4.13 Listing 4.14
  2. Constraints on SA configuration, Listing 4.15
  3. Constraints on IPsec and TLS capabilities, Listing 4.17

4.2.2 Protection Requirements Related Constraints

As mentioned in section 4.2, there is another important group of constraints yet to be discussed, which pertains to the protection requirements themselves. These constraints are generated within the *Checker* class. Specifically, the `formalizeRequirements()` method in the `VerefooProxy` module calls the `createRequirementConstraints()` method of the *Checker* module (as shown in Figure 4.1).

The `createRequirementConstraints` method is invoked for each security requirement specified by the user in the input XML file, serving as a wrapper for other three methods specifically designed for the different types of security requirement: `createIsolationConstraints()`, `createReachabilityConstraints()`, and `createProtectionConstraints()`.

Since the first two are used exclusively when VEREFOO configures firewalls, this thesis will focus its analysis only on the Protection Requirement type, which involves the establishment of a VPN tunnel for enforcement.

The `createRequirementConstraints` method operates at the granularity of individual security requirements; each security requirement is associated with multiple flows, and each flow consists of nodes and links. The method iterates over each flow within the security requirement and then over each node and link in each flow, generating constraints for:

1. Untrusted nodes
2. Inspector and Destination nodes
3. Nodes with VPN capabilities
4. Untrusted links

Untrusted Nodes Handling

```
1  if(un.contains(node.getNode().getName()) ) {
2      List<AllocationNode> predecessors= new ArrayList<>();
3      int unPos=flow.getPath().getNodes().indexOf(node);
4      for(int i=0;i<flow.getPath().getNodes().size(); i++) {
5          if(i<unPos) {
6              predecessors.add(flow.getPath().getNodes().get(i));
7          }
8      }
9      List<BoolExpr> protectExprs = new ArrayList<>();
10     List<BoolExpr> disProtectExprs = new ArrayList<>();
11     List<BoolExpr> constraints= new ArrayList<>();
12     for(AllocationNode predecessor: predecessors) {
13         protectExprs.add((BoolExpr)nctx.protect.apply(
14             predecessor.getZ3Name(), ctx.mkInt(flow.getIdFlow())) );
15         disProtectExprs.add((BoolExpr)
               nctx.disProtect.apply(predecessor.getZ3Name(),
               ctx.mkInt(flow.getIdFlow())));
16     }
17     BoolExpr[] bl2 = new BoolExpr[disProtectExprs.size()];
18     disProtectExprs.toArray(bl2);
19     int z=0;
20     int i=0;
21
22     for(BoolExpr boolProtect: protectExprs){
23         BoolExpr[] disProtect= new
               BoolExpr[disProtectExprs.size()-(i)];
24         disProtect[z]=boolProtect;
25         z++;
26         for (int j=i+1; j<protectExprs.size();j++){
27             disProtect[z]=(ctx.mkEq(bl2[j],ctx.mkFalse()));
28             z++;
29         }
30         i++;
31         z=0;
32         constraints.add(ctx.mkAnd(disProtect));
33     }
34     BoolExpr[] finalConstraints = new BoolExpr[constraints.size()];
35     constraints.toArray(finalConstraints);
```

```
36     untrustedConstraints.add(ctx.mkOr(finalConstraints));
37 }
```

Listing 4.18: Constraints on Untrusted Nodes

If the list of untrusted nodes `un` of the current traffic flow contains the node under analysis:

- **Lines 2-8**: The list `predecessors` is computed, containing all the nodes preceding the untrusted one.
- **Lines 9-16**: For each node in the `predecessors` list, a pair of boolean expressions is created and stored in two lists, `protectExprs` and `disProtectExprs`.
  - `protectExprs`: Contains boolean expressions stating that the node applies protection to the specific traffic flow.
  - `disProtectExprs`: Contains boolean expressions stating that the node removes protection from the specific traffic flow.
- **Lines 22-33**: For each boolean expression in `protectExprs`, a constraint is generated by concatenating the `boolProtect` expression with the subsequent `disProtect` expressions (with their value forced to `false`) using the *AND* logical operator. The meaning of this constraint is interpreted as: "If the node applies protection to the traffic flow, none of the subsequent nodes (preceding the untrusted node) should remove that protection."
- **Lines 34-36**: All the generated constraints are combined into a single constraint using the *OR* operator.

The result of this algorithm is that if a node in a traffic flow associated with the Protection Requirement is declared as untrusted, one of the preceding nodes must apply protection to the traffic flow, and that protection must remain in place until the untrusted node is traversed.

Inspector and Destination Nodes Handling

These two types of nodes, though serving different purposes, share a common requirement: the traffic that reaches them must not be protected. In the case of an inspector node, this is because if the traffic is encrypted, the node would be unable to perform any inspection. Similarly, for the destination node, it would be unable to process the data packets if any form of protection remains applied. Therefore, it is crucial to ensure that the network traffic reaching these nodes is unprotected and in clear form.

```
1 if(node.getNode().getName().equals(sr.getOriginalProperty().getDst())
     || in.contains(node.getNode().getName())) {
2     List<AllocationNode> predecessors= new ArrayList<>();
3     int dstPos=flow.getPath().getNodes().indexOf(node);
4     for(int i=0;i<flow.getPath().getNodes().size(); i++) {
```

```
5         if(i<=dstPos) {
6             predecessors.add(flow.getPath().getNodes().get(i));
7         }
8     }
9     List<BoolExpr> protectExprs = new ArrayList<>();
10    List<BoolExpr> disProtectExprs = new ArrayList<>();
11    List<BoolExpr> constraints= new ArrayList<>();
12    for(AllocationNode predecessor: predecessors) {
13        protectExprs.add((BoolExpr)nctx.protect.apply(
14            predecessor.getZ3Name(), ctx.mkInt(flow.getIdFlow())) );
15        disProtectExprs.add((BoolExpr)
              nctx.disProtect.apply(predecessor.getZ3Name(),
              ctx.mkInt(flow.getIdFlow())));
16    }
17    BoolExpr[] bl2 = new BoolExpr[disProtectExprs.size()];
18    disProtectExprs.toArray(bl2);
19    int z=0;
20    int i=0;
21    for(BoolExpr boolProtect: protectExprs){
22        BoolExpr[] disProtect= new
              BoolExpr[disProtectExprs.size()-(i+1)];
23        for (int j=i+1; j<protectExprs.size();j++){
24            disProtect[z]=bl2[j];
25            z++;
26        }
27        z=0;
28        constraints.add(ctx.mkImplies(boolProtect,ctx.mkOr(disProtect)));
29        i++;
30    }
31    BoolExpr[] finalConstraints = new BoolExpr[constraints.size()];
32    constraints.toArray(finalConstraints);
33    inspectorConstraints.add(ctx.mkAnd(finalConstraints));
34 }
```

Listing 4.19: Constraints on Inspector and Destination nodes

If the list of inspector nodes `in` in the current traffic flow contains the node under analysis, or the node under analysis is the destination of the traffic:

- **Lines 2-16**: As in Listing 4.18, the list of nodes preceding the current node is computed, and the list of boolean expressions regarding the application or removal of protection on the traffic flow is created.
- **Lines 17-30**: For each `boolProtect` expression, a constraint is generated by combining the protection expression with the subsequent `disProtect` expressions (concatenated using the *OR* logical operator) through the *IMPLY* logical operator. The meaning of this constraint is: "If the node applies protection to the traffic flow, then one of the subsequent nodes (preceding the inspector or destination node) should remove that protection."

- **Lines 31-34**: Once again, all the generated constraints are combined into a single constraint using the *OR* operator.

The result of this algorithm is that if a node in a traffic flow associated with the Protection Requirement is declared as an inspector or serves as the destination, then if any preceding node applies protection to the traffic flow, another node between the one applying the protection and the inspector/destination node must remove that protection.

Nodes with VPN Capabilities

The following algorithm verifies that a node possessing VPN capabilities (so it is able to apply or remove protection) also support the security technologies and the corresponding capabilities.

```
1  BoolExpr b1_1=ctx.mkEq(nctx.protect.apply(node.getZ3Name(),
       ctx.mkInt(flow.getIdFlow())), ctx.mkTrue() );
2  BoolExpr b1_2=ctx.mkEq(nctx.disProtect.apply(node.getZ3Name(),
       ctx.mkInt(flow.getIdFlow())), ctx.mkTrue() );
3  BoolExpr b1= ctx.mkOr(b1_1,b1_2);
4  HashMap<String, ArrayList<String>> tecCapMap = new HashMap<>();
5  for(SecurityTechnologyType st : sts) {
6      ArrayList<String> caps = new ArrayList<>();
7      String string = st.toString()+"_" + encAlg;
8      String string2 = st.toString()+"_" + authAlg;
9      caps.add(string);
10     caps.add(string2);
11     tecCapMap.put(st.toString(), caps);
12 }
13 List<BoolExpr> tecConstraints = new ArrayList<>();
14 for (Map.Entry<String,ArrayList<String>> entry :
       tecCapMap.entrySet()) {
15     List<BoolExpr> capConstraints = new ArrayList<>();
16     for(String c : entry.getValue()) {
17         BoolExpr b2 =
               ctx.mkEq(nctx.supportCap.apply(node.getZ3Name()
               ,ctx.mkInt(flow.getIdFlow()), ctx.mkString(c) ),
               ctx.mkTrue());
18         capConstraints.add(b2);
19     }
20     BoolExpr[] arraycapConstraints = new
           BoolExpr[capConstraints.size()];
21     BoolExpr finalCapConstraint =
           ctx.mkAnd(capConstraints.toArray(arraycapConstraints));
22     tecConstraints.add(finalCapConstraint);
23     }
24 BoolExpr[] arraytecConstraints = new
       BoolExpr[tecConstraints.size()];
```

```
25 BoolExpr finalTecConstraint =
      ctx.mkOr(tecConstraints.toArray(arraytecConstraints));
26 BoolExpr capConstraint1 = ctx.mkImplies(b1, finalTecConstraint);
27 constraintList.add(capConstraint1);
```

Listing 4.20: Constraints on VPN capabilities by Checker

- **Lines 1-3**: The Boolean expression `b1` is defined, and it is `true` if the node either applies protection or removes protection (dis-protection) on the traffic flow.
- **Lines 4-12**: A HashMap called `tecCapMap` is created, where the key is the security technology (i.e., IPsec and TLS), and the value is an array containing strings that represent the required authentication and encryption algorithms (e.g., "IPSEC_SHA256").
- **Lines 13-23**: For each security technology:
  - **Lines 14-18**: For each capability associated with the security technology (both the encryption and authentication algorithms), a constraint is generated. This constraint asserts that the node must support the corresponding security capability, and it is inserted into `capConstraints`.
  - **Lines 20-22**: The constraints in the `capConstraints` list are combined into a single constraint `finalCapConstraint` using a logical *AND* operator. This means that for a given security technology, all the necessary capabilities must be supported by the node.
- **Lines 24-27**: The elements of the `tecConstraints` list (which contains the constraints related to the two security technologies, IPsec and TLS) are combined using a logical *OR* operator. This implies that the node must support the necessary capabilities for at least one of the two security technologies. Finally, an *IMPLY* operator is used to logically connect the protection/dis-protection condition with the security technology support.

The result of this algorithm enforces the following constraint: "If a node applies or removes protection on a traffic flow, it must also support all the necessary capabilities for at least one security technology."

Untrusted Links Handling

The algorithm that generates constraints for untrusted links is nearly identical to the one discussed in Listing 4.18. The only difference lies in how the predecessor nodes are identified, as a link is a logical connection between a pair of nodes. In this case, the `predecessors` list contains all the nodes preceding the destination of the untrusted link. Otherwise, the algorithm follows the same steps, ensuring that if a node applies protection to the traffic, no subsequent preceding node removes that protection, thereby guaranteeing that traffic remains protected when passing through untrusted links in the network.

4.3 Z3 Model Analysis

All the generated constraints discussed in section 4.2 are submitted by the Checker module to the framework's embedded Z3 Solver, the MaxSMT solver developed by Microsoft, which resolves the MaxSMT problem using the `Check()` method. The result of this operation is the generation of the Z3 Model, consisting of a set of strings that satisfy all hard constraints and maximize the sum of the weights assigned to the satisfiable soft constraints. These strings specifically represent the previously defined free variables and the values assigned to them by the Z3 solver.

To provide a clear and straightforward example of the Z3 Model, a specific XML input file has been created:

```
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
    <graphs>
        <graph id="0">
            <node functional_type="WEBCLIENT" name="10.0.0.1">
                <neighbour name="50.0.0.1" />
            </node>
            <node name="50.0.0.1">
                <neighbour name="10.0.0.1" />
                <neighbour name="30.0.0.1" />
            </node>
            <node functional_type="FORWARDER" name="30.0.0.1">
                <neighbour name="50.0.0.1" />
                <neighbour name="50.0.0.2" />
            </node>
            <node name="50.0.0.2">
                <neighbour name="30.0.0.1" />
                <neighbour name="20.0.0.1" />
            </node>
            <node functional_type="WEBSERVER" name="20.0.0.1">
                <neighbour name="50.0.0.2" />
            </node>
        </graph>
    </graphs>
    <Constraints>
        <NodeConstraints/>
        <LinkConstraints />
    </Constraints>
    <PropertyDefinition>
        <Property graph="0" name="ProtectionProperty" src="10.0.0.1"
dst="20.0.0.1">
            <protectionInfo>
                <untrustedNode node="30.0.0.1"></untrustedNode>
            </protectionInfo>
        </Property>
    </PropertyDefinition>
</NFV>
```

Listing 4.21: Simple Input Testcase

This input file describes the following topology:

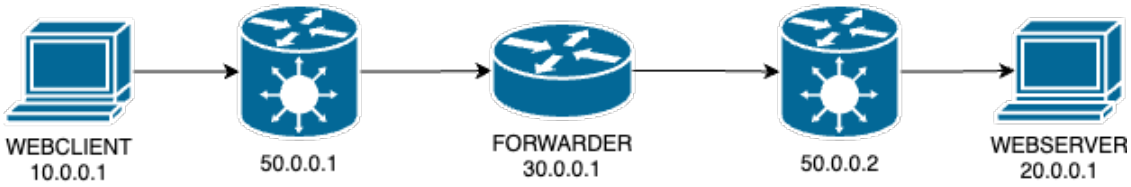


Figure 4.2: Simple VPN Topology

In this topology, the *FORWARDER* node with IP addresses setted to 30.0.0.1 is declared **untrusted** in the `Property` section of the input file, so on the node with IP address 50.0.0.1 a *VPN_GATEWAY* with *ACCESS* behavior should be deployed, while on the node with IP address 50.0.0.2 a *VPN_GATEWAY* with *EXIT* behavior is expected to be configured.

The corresponding Z3 model generated by the framework, which strings have been reordered to enhance clarity and understanding of the various sections, is the following:

```
1     (define-fun null () PortRange (port_range_constructor 0 65535))
2     (define-fun null () Address (ip_constructor 0 0 0 0))
3     (define-fun wildcard () Address (ip_constructor (- 1) (- 1) (-
         1) (- 1)))
4
5     (define-fun |10.0.0.1| () Address (ip_constructor 10 0 0 1))
6     (define-fun |50.0.0.1| () Address (ip_constructor 50 0 0 1))
7     (define-fun |30.0.0.1| () Address (ip_constructor 30 0 0 1))
8     (define-fun |50.0.0.2| () Address (ip_constructor 50 0 0 2))
9     (define-fun |20.0.0.1| () Address (ip_constructor 20 0 0 1))
10
11    (define-fun |\|10.0.0.1\|_vpnCapIsUsed| () Bool false)
12
13    (define-fun |\|50.0.0.1\|_used| () Bool true)
14    (define-fun |\|50.0.0.1\|used0| () Bool true)
15    (define-fun |\|50.0.0.1\|_auto_src_0| () Address
         (ip_constructor 10 0 0 1))
16    (define-fun |\|50.0.0.1\|_auto_dst_0| () Address
         (ip_constructor 20 0 0 1))
17    (define-fun |\|50.0.0.1\|_auto_srcp_0| () PortRange
         (port_range_constructor 0 65535))
18    (define-fun |\|50.0.0.1\|_auto_dstp_0| () PortRange
         (port_range_constructor 0 65535))
19    (define-fun |\|50.0.0.1\|_auto_proto_0| () Int 0)
20    (define-fun |\|50.0.0.1\|_auto_src_ip_4_0| () Int 1)
21    (define-fun |\|50.0.0.1\|_auto_src_ip_3_0| () Int 0)
22    (define-fun |\|50.0.0.1\|_auto_src_ip_2_0| () Int 0)
23    (define-fun |\|50.0.0.1\|_auto_src_ip_1_0| () Int 10)
24    (define-fun |\|50.0.0.1\|_auto_dst_ip_4_0| () Int 1)
25    (define-fun |\|50.0.0.1\|_auto_dst_ip_3_0| () Int 0)
```

```
26    (define-fun |\|50.0.0.1\|_auto_dst_ip_2_0| () Int 0)
27    (define-fun |\|50.0.0.1\|_auto_dst_ip_1_0| () Int 20)
28
29    (define-fun |\|50.0.0.2\|_used| () Bool true)
30    (define-fun |\|50.0.0.2\|used0| () Bool true)
31    (define-fun |\|50.0.0.2\|_auto_src_0| () Address
         (ip_constructor 10 0 0 1))
32    (define-fun |\|50.0.0.2\|_auto_dst_0| () Address
         (ip_constructor 20 0 0 1))
33    (define-fun |\|50.0.0.2\|_auto_srcp_0| () PortRange
         (port_range_constructor 0 65535))
34    (define-fun |\|50.0.0.2\|_auto_dstp_0| () PortRange
         (port_range_constructor 0 65535))
35    (define-fun |\|50.0.0.2\|_auto_proto_0| () Int 0)
36    (define-fun |\|50.0.0.2\|_auto_src_ip_4_0| () Int 1)
37    (define-fun |\|50.0.0.2\|_auto_src_ip_3_0| () Int 0)
38    (define-fun |\|50.0.0.2\|_auto_src_ip_2_0| () Int 0)
39    (define-fun |\|50.0.0.2\|_auto_src_ip_1_0| () Int 10)
40    (define-fun |\|50.0.0.2\|_auto_dst_ip_4_0| () Int 1)
41    (define-fun |\|50.0.0.2\|_auto_dst_ip_3_0| () Int 0)
42    (define-fun |\|50.0.0.2\|_auto_dst_ip_2_0| () Int 0)
43    (define-fun |\|50.0.0.2\|_auto_dst_ip_1_0| () Int 20)
44
45    (define-fun |\|20.0.0.1\|_vpnCapIsUsed| () Bool false)
46
47    (define-fun |0_auto_tunnelSrc_0| () Address (ip_constructor 10
         0 0 1))
48    (define-fun |0_auto_tunnelDst_0| () Address (ip_constructor 20
         0 0 1))
49    (define-fun |0_auto_tunnelSrc_1| () Address (ip_constructor 10
         0 0 1))
50    (define-fun |0_auto_tunnelDst_1| () Address (ip_constructor 20
         0 0 1))
51    (define-fun |0_auto_tunnelSrc_2| () Address (ip_constructor 50
         0 0 1))
52    (define-fun |0_auto_tunnelDst_2| () Address (ip_constructor 50
         0 0 2))
53    (define-fun |0_auto_tunnelSrc_3| () Address (ip_constructor 50
         0 0 1))
54    (define-fun |0_auto_tunnelDst_3| () Address (ip_constructor 50
         0 0 2))
55    (define-fun |0_auto_tunnelSrc_4| () Address (ip_constructor 10
         0 0 1))
56    (define-fun |0_auto_tunnelDst_4| () Address (ip_constructor 20
         0 0 1))
57
58    (define-fun nodeHasAddr ((x!0 Node) (x!1 Address)) Bool
59      (let ((a!1 (ite (= x!1 (ip_constructor 50 0 0 2))
```

```
60                    (ip_constructor 50 0 0 2)
61                    (ite (= x!1 (ip_constructor 20 0 0 1))
62                         (ip_constructor 20 0 0 1)
63                         (ip_constructor 3 30 30 30)))))
64       (let ((a!2 (ite (= x!1 (ip_constructor 50 0 0 1))
65                    (ip_constructor 50 0 0 1)
66                    (ite (= x!1 (ip_constructor 10 0 0 1))
67                         (ip_constructor 10 0 0 1)
68                         a!1))))
69       (let ((a!3 (ite (= x!1 (ip_constructor 30 0 0 1))
70                    (ip_constructor 30 0 0 1)
71                    a!2)))
72         (or (and (= x!0 |50.0.0.2|) (= a!3 (ip_constructor 50 0 0
              2)))
73             (and (= x!0 |50.0.0.1|) (= a!3 (ip_constructor 50 0 0
                 1)))
74             (and (= x!0 |20.0.0.1|) (= a!3 (ip_constructor 20 0 0
                 1)))
75             (and (= x!0 |30.0.0.1|) (= a!3 (ip_constructor 30 0 0
                 1)))
76             (and (= x!0 |10.0.0.1|) (= a!3 (ip_constructor 10 0 0
                 1)))))))
77     (define-fun supportCap ((x!0 Node) (x!1 Int) (x!2 String)) Bool
          true)
78     (define-fun deny ((x!0 Node) (x!1 Int)) Bool false)
79     (define-fun addrToNode ((x!0 Address)) Node
80       (ite (= x!0 (ip_constructor 50 0 0 2)) |50.0.0.2|
81       (ite (= x!0 (ip_constructor 30 0 0 1)) |30.0.0.1|
82       (ite (= x!0 (ip_constructor 10 0 0 1)) |10.0.0.1|
83       (ite (= x!0 (ip_constructor 20 0 0 1)) |20.0.0.1|
84         |50.0.0.1|)))))
85
86     (define-fun disProtect ((x!0 Node) (x!1 Int)) Bool
87       (and (= x!0 |50.0.0.2|) (= x!1 0)))
88     (define-fun protect ((x!0 Node) (x!1 Int)) Bool
89       (and (= x!0 |50.0.0.1|) (= x!1 0)))
```

Listing 4.22: Simple Z3 Model

- **Lines 1-3**: Wildcards and null values for ports and IP addresses are defined.
- **Lines 5-9**: IP addresses of the nodes in the topology are specified.
- **Lines 11-45**: Free variables for the VPN Gateways and EndHosts are defined.
  - **Line 11**: The free variable `_vpnCapIsUsed` for the *WEBCLIENT* node is set to `false`, as the input XML file did not specify VPN capabilities for this node.

  - **Lines 13-27**: Free variables for the node with IP address 50.0.0.1 are defined.
    - **Line 13**: The free variable `_used` is set to `true`, indicating that the node 50.0.0.1 is used as a *VPN_GATEWAY*.
    - **Line 14**: The free variable `used0` is set to `true`, meaning the node is used as a *VPN_GATEWAY* for the flow with `id=0`.
    - **Lines 15-27**: The free variables related to the placeholder rules described in Listing 4.2 are assigned their respective values.
  - **Lines 29-43**: Similar to lines 13-27, but for the node with IP address 50.0.0.2.
  - **Line 45**: Same as for the *WEBCLIENT*, but applicable to the *WEBSERVER*.

- **Lines 47-56**: Free variables related to the VPN tunnels are assigned the source and destination IP addresses for each node in the flow path, constrained by the conditions set in Listing 4.7, Listing 4.10, Listing 4.11.

- **Lines 58-84**: Two functions with complementary purposes are defined: `nodeToAddr` maps each node's name to its corresponding IP address, while `addrToNode` performs the reverse mapping.

- **Lines 86-87**: Lists the nodes that remove protection and the traffic flows on which they act.

- **Lines 88-89**: Lists the nodes that apply protection and the traffic flows on which they act.

The two most significant sections of the Z3 model pertain to VPN tunnels and the application or removal of protection by nodes.

For the VPN tunnels, the structure of each string follows this format: `flowId + "_auto_tunnelSrc/Dst" + pos`. Here, `flowId` is the integer assigned to each flow, identifying it uniquely, while `pos` indicates the node's position within the flow path. For example,
`(define-fun |0_auto_tunnelSrc_0| () Address (ip_constructor 10 0 0 1))` specifies that the tunnel source IP address for the first node of the flow with `id=0` is 10.0.0.1. Given the simplicity of the input SG topology, only a single flow with `id=0` is processed. This allows for an easy observation that the tunnel source and destination IP addresses initially match the actual traffic addresses (10.0.0.1 and 20.0.0.1, respectively) until the untrusted node is reached (`pos=2`). At this point, the tunnel addresses switch to those of the VPN gateways (50.0.0.1 and 50.0.0.2, respectively). Upon exiting the VPN tunnel, the addresses revert to those of the original traffic flows.

The other crucial section of the Z3 model is the final part, where nodes listed in `protect` apply protection to the traffic identified by an integer, and they will be assigned an *ACCESS* behavior when the model is translated into the XML output file. Similarly, nodes in the `disProtect` list remove protection from the traffic identified by the integer, and they will be assigned an *EXIT* behavior.

4.4 XML Output File Analysis

After generating the Z3 model, the Translator module, as described in Figure 4.1, parses the model and produces an XML file in which the VPN Gateways are allocated and configured with the necessary Security Associations (SAs). The output XML file follows a schema similar to the one described in Listing 4.1, allowing the user to be familiar with its structure and easily use it to deploy and configure the security services specified in the output.

To provide an example of the output XML file coherent to the analysis performed in the previous section, the same simple VPN topology and input XML file referenced in Listing 4.21 has been used. The framework produced the following output:

```
<NFV xsi:noNamespaceSchemaLocation="./xsd/nfvSchema.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <graphs>
        <graph id="0">
            <node name="10.0.0.1" functional_type="WEBCLIENT">
                <neighbour name="50.0.0.1"/>
            </node>
            <node name="50.0.0.1" functional_type="VPNGateway">
                <neighbour name="10.0.0.1"/>
                <neighbour name="30.0.0.1"/>
                <configuration name="AutoConf">
                    <vpnGateway>
                        <securityAssociation>
                            <behavior>ACCESS</behavior>
                            <startChannel></startChannel>
                            <endChannel></endChannel>
                            <source>10.0.0.1</source>
                            <destination>20.0.0.1</destination>
                            <protocol>ANY</protocol>
                            <src_port>*</src_port>
                            <dst_port>*</dst_port>
                        </securityAssociation>
                    </vpnGateway>
                </configuration>
            </node>
            <node name="30.0.0.1" functional_type="FORWARDER">
                <neighbour name="50.0.0.1"/>
                <neighbour name="50.0.0.2"/>
            </node>
            <node name="50.0.0.2" functional_type="VPNGateway">
                <neighbour name="30.0.0.1"/>
                <neighbour name="20.0.0.1"/>
                <configuration name="AutoConf">
                    <vpnGateway>
                        <securityAssociation>
                            <behavior>EXIT</behavior>
                            <startChannel></startChannel>
                            <endChannel></endChannel>
                            <source>10.0.0.1</source>
                            <destination>20.0.0.1</destination>
                            <protocol>ANY</protocol>
                            <src_port>*</src_port>
```

```
42                              <dst_port>*</dst_port>
43                          </securityAssociation>
44                      </vpnGateway>
45                  </configuration>
46              </node>
47              <node name="20.0.0.1" functional_type="WEBSERVER">
48                  <neighbour name="50.0.0.2"/>
49              </node>
50          </graph>
51      </graphs>
52      <Constraints>
53          <NodeConstraints/>
54          <LinkConstraints/>
55      </Constraints>
56      <PropertyDefinition>
57          <Property name="ProtectionProperty" graph="0" src="10.0.0.1"
    dst="20.0.0.1" lv4proto="ANY" src_port="null" dst_port="null" isSat="true">
58              <protectionInfo>
59                  <untrustedNode node="30.0.0.1"/>
60              </protectionInfo>
61          </Property>
62      </PropertyDefinition>
63      <profile name="hybrid"/>
64  </NFV>
```

Listing 4.23: Simple VEREFOO Output

As expected, the nodes with addresses `50.0.0.1` and `50.0.0.2` have been configured as **VPNGateway**. Specifically, each node has been set up with a Security Association: the first node is assigned an *ACCESS* behavior, while the second is assigned an *EXIT* behavior, consistent with the previously analyzed Z3 model. Since the Security Property specified in the input did not define specific ports or a Layer 4 protocol, wildcard values have been assigned (`ANY` for the protocol and `*` for the ports).

In the `PropertyDefinition` section of the output, the property has been enriched with additional fields not present in the input file: `lv4proto`, `src_port`, and `dst_port` are now populated with the values of the configured SAs, while the field `isSat` is assigned a boolean value indicating whether the configured SAs satisfy the specified property.

Chapter 5

Enhancements and Performance Evaluation

5.1 Identified Issues and Limitations of the Framework

During the framework analysis, conducted primarily through detailed debugging and a careful examination of the Z3 model and the produced output, several issues were identified. These issues can be classified as follows:

- Configuration and Pathing Errors
- Output Quality and Efficiency Issue
- Optimization Deficiencies

5.1.1 Configuration and Pathing Errors

The defects in this category impact both flow path generation and the configuration of VPN Gateways through constraints. In this context, three specific issues have been identified:

1. EndHosts that do not participate in the traffic flow are incorrectly included in the flow path.
2. Incorrect weights are assigned to soft constraints.
3. Protection constraints for inspector and destination nodes are constructed erroneously.

The first issue was identified while analyzing flow paths in topologies with multiple *WEBCLIENTS* and *WEBSERVERS*. Specifically, when the framework computes a flow path for a particular Security Requirement, EndHosts that are not

relevant to the Security Requirement in question are still included in the flow path. For example, consider the following network topology:

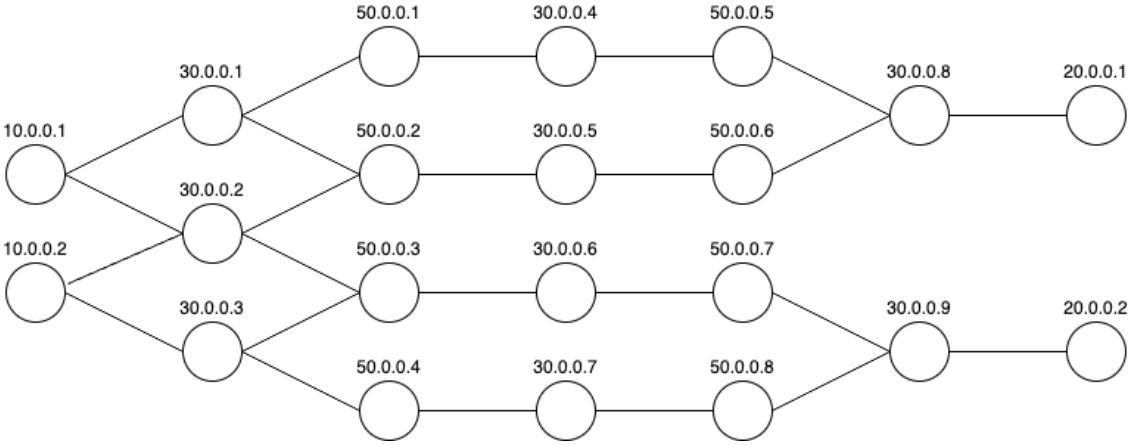
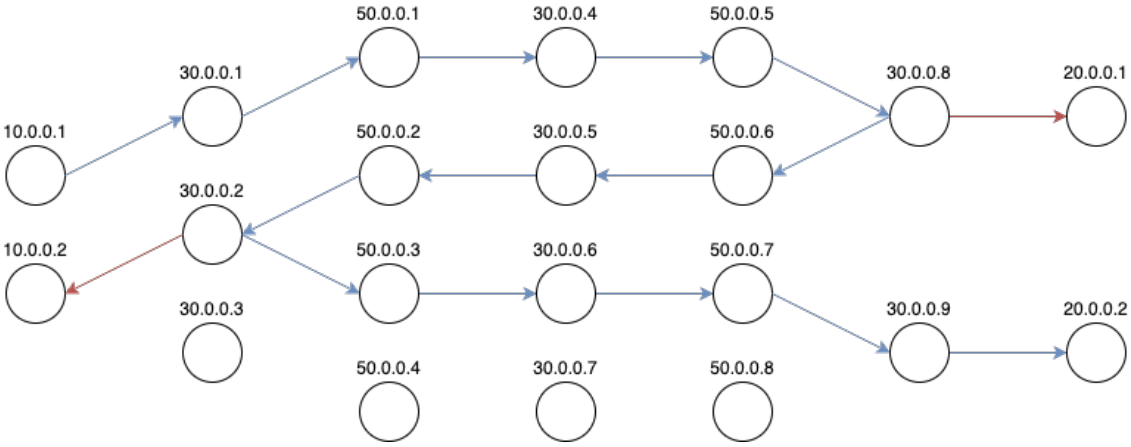


Figure 5.1: Sample topology with multiple clients and servers

If a Security Requirement requires protection on the traffic with `src=10.0.0.1` and `dst=20.0.0.2`, one of the flow paths computed by the framework is the following:



The blue arrows indicate the correct sequence of nodes in the flow paths, while the red arrows highlight points where the recursive algorithm responsible for generating the flow path encounters a "sink" and cannot proceed further. This occurs because EndHosts, such as 10.0.0.2 and 20.0.0.1, lack forwarding capabilities, causing the algorithm to backtrack to the previous step and attempt an alternative route. However, even though the two EndHosts with IP addresses 10.0.0.2 and 20.0.0.1 are recognized as not being part of the flow path leading to the Security Requirement's destination, they are not removed from the path. This affects subsequent constraint generation, as these nodes are incorrectly considered part of the path.

Regarding the issue of incorrect weight assignments to soft constraints, it is helpful to revisit the code snippets provided in Listing 4.13 for a clearer understanding. The soft constraints within the lists `softConstrWildcard`, `softConstrProtoWildcard`,

and `softConstrPorts` are assigned weights in the framework's NetContext module as follows:

```
1    softConstrWildcard.forEach(t->solver.AssertSoft(t._1,
        WIPWILDCARD, t._2));
2    softConstrProtoWildcard.forEach(t->solver.AssertSoft(t._1,
        WPROTOWILDCARD, t._2));
3    softConstrPorts.forEach(t->solver.AssertSoft(t._1, WPORTS,
        t._2));
```

Each element in these lists is submitted to the Z3 solver using the `solver.AssertSoft()` method, where the first argument is the constraint, the second argument specifies the assigned weight, and the third argument indicates the group to which the constraint belongs. The weights assigned to these soft constraints are as follows:

- `WIPWILDCARD`: 10
- `WPORTS`: -10
- `WPROTOWILDCARD`: -10

However, by examining the construction of these soft constraints, it becomes apparent that their effect is contrary to the intended outcome. By assigning a negative weight to the soft constraints related to ports and protocols, the solver is encouraged to prioritize assigning specific values over wildcard values. A similar issue occurs with the soft constraint related to the assignment of wildcard values for the source and destination IP addresses, where the solver is inclined to assign specific IP addresses rather than wildcard values.

Finally, an issue has been identified in the construction of constraints related to inspector and destination nodes. Specifically, in the algorithm described in Listing 4.19, if the last node in the `predecessors` list applies protection to the traffic, there is no further opportunity to remove that protection before the traffic reaches the inspector or destination node, thereby preventing them from functioning as intended. This issue also impacts the Z3 model, as an incomplete constraint is generated, which can potentially affect the values assigned to the free variables by the solver.

5.1.2 Output Quality and Efficiency Issue

In section 4.4, the framework's output was discussed, with an example provided using a very simple input. However, when the framework is supplied with a slightly more complex input file, such as the one describing the network topology illustrated in Figure 5.1, the output contains a significant amount of redundant information. For example, with this topology, if the user requests protection between each client-server pair, four Security Associations per VPN Gateway would be sufficient (10.0.0.1-20.0.0.1, 10.0.0.1-20.0.0.2, 10.0.0.2-20.0.0.1, 10.0.0.2-20.0.0.2). In practice, however, the framework's output contains many more SAs than necessary, often with repeated entries. For instance, the node with IP address 50.0.0.2 has been configured with eleven SAs:

```
...

<node name="50.0.0.2" functional_type="VPNGateway">
<neighbour name="30.0.0.1"/>
<neighbour name="30.0.0.2"/>
<neighbour name="30.0.0.5"/>
<configuration name="AutoConf">
    <vpnGateway>
        <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel></startChannel>
            <endChannel></endChannel>
            <source>10.0.0.2</source>
            <destination>20.0.0.1</destination>
            <protocol>ANY</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
        </securityAssociation>
        <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel></startChannel>
            <endChannel></endChannel>
            <source>10.0.0.1</source>
            <destination>20.0.0.1</destination>
            <protocol>ANY</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
        </securityAssociation>
        <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel></startChannel>
            <endChannel></endChannel>
            <source>10.0.0.1</source>
            <destination>20.0.0.2</destination>
            <protocol>ANY</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
        </securityAssociation>
        <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel></startChannel>
            <endChannel></endChannel>
            <source>10.0.0.1</source>
            <destination>20.0.0.1</destination>
            <protocol>ANY</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
        </securityAssociation>
        <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel></startChannel>
            <endChannel></endChannel>
            <source>10.0.0.2</source>
            <destination>20.0.0.1</destination>
            <protocol>ANY</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
```

```
58              </securityAssociation>
59              <securityAssociation>
60                  <behavior>ACCESS</behavior>
61                  <startChannel></startChannel>
62                  <endChannel></endChannel>
63                  <source>10.0.0.2</source>
64                  <destination>20.0.0.1</destination>
65                  <protocol>ANY</protocol>
66                  <src_port>*</src_port>
67                  <dst_port>*</dst_port>
68              </securityAssociation>
69              <securityAssociation>
70                  <behavior>ACCESS</behavior>
71                  <startChannel></startChannel>
72                  <endChannel></endChannel>
73                  <source>10.0.0.2</source>
74                  <destination>20.0.0.1</destination>
75                  <protocol>ANY</protocol>
76                  <src_port>*</src_port>
77                  <dst_port>*</dst_port>
78              </securityAssociation>
79              <securityAssociation>
80                  <behavior>ACCESS</behavior>
81                  <startChannel></startChannel>
82                  <endChannel></endChannel>
83                  <source>10.0.0.1</source>
84                  <destination>20.0.0.2</destination>
85                  <protocol>ANY</protocol>
86                  <src_port>*</src_port>
87                  <dst_port>*</dst_port>
88              </securityAssociation>
89              <securityAssociation>
90                  <behavior>ACCESS</behavior>
91                  <startChannel></startChannel>
92                  <endChannel></endChannel>
93                  <source>10.0.0.2</source>
94                  <destination>20.0.0.1</destination>
95                  <protocol>ANY</protocol>
96                  <src_port>*</src_port>
97                  <dst_port>*</dst_port>
98              </securityAssociation>
99              <securityAssociation>
100                 <behavior>ACCESS</behavior>
101                 <startChannel></startChannel>
102                 <endChannel></endChannel>
103                 <source>10.0.0.1</source>
104                 <destination>20.0.0.1</destination>
105                 <protocol>ANY</protocol>
106                 <src_port>*</src_port>
107                 <dst_port>*</dst_port>
108             </securityAssociation>
109             <securityAssociation>
110                 <behavior>ACCESS</behavior>
111                 <startChannel></startChannel>
112                 <endChannel></endChannel>
113                 <source>10.0.0.2</source>
114                 <destination>20.0.0.1</destination>
```

```
115                 <protocol>ANY</protocol>
116                 <src_port>*</src_port>
117                 <dst_port>*</dst_port>
118             </securityAssociation>
119         </vpnGateway>
120     </configuration>
121 </node>
122 ...
```

The redundancy not only affects the overall quality of the output produced by the framework but also leads to a considerable waste of resources in terms of memory usage. This can be especially problematic if a user decides to apply the generated configuration directly to a real network without manually removing the redundant SAs, potentially impacting network performance and efficiency.

5.1.3 Optimization Deficiencies

The issues in this category do not prevent the framework from functioning as intended, nor do they affect the correctness of the computed solution. However, they can impact the framework's performance and scalability to varying degrees, depending on the size of the network topology and the number of security requirements provided as input.

The analysis conducted in this thesis revealed two areas for improvement in the framework: the suboptimal generation of constraints related to protection requirements, and the absence of a pruning function to minimize the number of placeholder rule variables that need to be defined.

To better illustrate where the suboptimal generation of constraints occurs, it is helpful to outline the theoretical formulas, deeply discussed in [34], that guides this process:

$$\begin{aligned}&\forall f \in \phi(p).\, \forall n_i \in \pi(f) \,|\, n_i \in p.W.NU.\\ &\quad \exists n_j \in \pi(f) \,|\, n_j \prec n_i.\, \big(\text{protect}_x(n_j, f) \wedge\\ &\quad \forall n_k \in \pi(f) \,|\, n_j \prec n_k \prec n_i.\, \neg\text{unprotect}_x(n_k, f)\big)\end{aligned} \tag{5.1}$$

$$\begin{aligned}&\forall f \in \phi(p).\, \forall n_i \in \pi(f) \,|\, n_i \in p.W.NI.\\ &\quad \forall n_j \in \pi(f) \,|\, n_j \prec n_i.\, \big(\text{protect}_c(n_j, f) \Rightarrow\\ &\quad \exists n_k \in \pi(f) \,|\, n_j \prec n_k \prec n_i.\, \text{unprotect}_c(n_k, f)\big)\end{aligned} \tag{5.2}$$

$$\begin{aligned}&\forall f \in \phi(p).\, \big(\exists n_i \in \pi(f).\, \text{protect}_x(n_i, f)\big) \Rightarrow\\ &\quad \big(\exists n_j \in \pi(f) \,|\, n_i \prec n_j.\, \text{unprotect}_x(n_j, f)\big)\end{aligned} \tag{5.3}$$

The formula (5.1) specifies that if a node n_i traversed by a flow f satisfying the security requirements is classified as untrustworthy, then there must be at least one node n_j preceding n_i in the flow path that enforces the required protection on the traffic flow, and this protection must remain intact, without being removed by any node n_k between n_j and n_i.

The formula (5.2) indicates that if a node n_i traversed by a flow f satisfying the security requirements is classified as an inspector, then f must not be protected by confidentiality when crossing n_i. Consequently, if a node n_j preceding n_i in the flow path enforces confidentiality on the traffic flow, this protection must be removed by a node n_k between n_j and n_i.

The formula (5.3) states that if a node n_i applies protection to a flow f, this protection must be removed by a subsequent node n_j in the path followed by f.

The three formulas have been fully implemented in Listing 4.18 and Listing 4.19; however, the implemented algorithms generate constraints for applying and removing protection on the traffic flow without considering the nature of the nodes and their actual capabilities. This oversight leads to the generation of numerous unnecessary constraints, making the Z3 model heavier and more computationally demanding to process.

The second suboptimization issue identified in the framework's source code is, as previously mentioned, the absence of a pruning function to minimize the number of placeholder rules that need to be defined. Specifically, while a method intended for this purpose, called `minimizePlaceholderRules()`, exists within the VPNGateway and EndHosts function modules, its implementation is as follows:

```
1     private int minimizePlaceholderRules() {
2     List<Flow> allProperties =
          source.getFlows().values().stream().collect(Collectors.toList());
3     int minimizedRules = allProperties.size();
4     return minimizedRules;
5  }
```

Listing 5.1: Current Implementation of pruning function

Essentially, this method merely collects all traffic flows crossing the node into a list and returns the size of this list as the minimum number of placeholder rules required. It is evident that this approach does not assist in computing the output by reducing the number of free variables the solver needs to consider, indicating that further development is necessary to optimize this aspect effectively.

5.2 Enhancements and Issues Addressing

To address the issues discussed previously, the source code of the responsible algorithms has been modified. In some instances, a simple modification was sufficient to resolve an issue, while in others, a more substantial reworking of the algorithm was necessary.

5.2.1 Addressing the Configuration and Pathing Errors

Starting from the pathing error, the algorithm responsible of the generation of the paths of nodes is the following:

```
1  private void recursivePathGeneration(List<List<AllocationNode>>
       allPaths, List<AllocationNode> currentPath, AllocationNode
       source, AllocationNode destination, AllocationNode current,
       Set<String> visited, int level) {
2      currentPath.add(level, current);
3      visited.add(current.getNode().getName());
4      List<Neighbour> listNeighbours =
           current.getNode().getNeighbour();
5      if(destination.getNode().getName().equals(current.getNode().getName()))
           {
6          List<AllocationNode> pathToStore = new ArrayList<>();
7          for(int i = 0; i < currentPath.size(); i++) {
8              pathToStore.add(i, currentPath.get(i));
9          }
10         allPaths.add(pathToStore);
11         visited.remove(current.getNode().getName());
12         currentPath.remove(level);
13         return;
14     }
15     if(level != 0) {
16         if(current.getNode().getFunctionalType() ==
               FunctionalTypes.WEBCLIENT ||
               current.getNode().getFunctionalType() ==
               FunctionalTypes.WEBSERVER) {
17             return;
18         }
19     }
20     for(Neighbour n : listNeighbours) {
21         if(!visited.contains(n.getName())) {
22             AllocationNode neighbourNode =
                   allocationNodes.get(n.getName());
23             level++;
24             recursivePathGeneration(allPaths, currentPath, source,
                   destination, neighbourNode, visited, level);
25             level--;
26         }
27
28     }
29     visited.remove(current.getNode().getName());
30     currentPath.remove(level);
31     return;
32 }
```

Listing 5.2: Recursive method for flow path generation

This method is invoked by the wrapper method `generateFlowPaths()` within the VerefooProxy module of the framework. The wrapper method calls `recursivePathGeneration()`, passing the source and destination addresses of the

security requirements for which the flow paths need to be constructed. As with any recursive algorithm, there is a base case that produces a result without further recursion. In this case, the base condition is that the currently analyzed node matches the destination of the security requirement. If this condition is met, the path is stored; otherwise, the algorithm recurses by analyzing the next neighboring node of the current one until it reaches the base case or encounters a node with a functional type of *WEBCLIENT* or *WEBSERVER*. The issue described in the previous section is rooted in this second condition: when a *WEBCLIENT* or *WEBSERVER* node is traversed by the algorithm, it is not removed from the `currentPath` list. As a result, even though the algorithm correctly backtracks to the previous recursion level and continues the path search, these nodes remain in the final path, where they are mistakenly considered as legitimate path nodes during constraint generation. By adding the line `currentPath.remove(level);` before the `return` instruction at line 17, the EndHost node is correctly removed from the list, ensuring it does not appear in the path.

Regarding errors related to configuration and constraint generation, an inconsistency between the intended framework behavior—specifically, assigning wildcard values to the free variables wherever possible—and the actual behavior was identified by examining the weights assigned to soft constraints. As mentioned previously, the sign of the values of the weights must be coherent with the way the associated soft constraints are constructed. For instance, the soft constraints specifying that wildcard values should be assigned to the Layer 4 protocol and ports utilize the `mkEq` boolean operator in Z3:

```
1 nctx.softConstrProtoWildcard.add(new Tuple<BoolExpr, String>(
2     (ctx.mkEq( proto, ctx.mkInt(0)) ),"vpn_auto_conf"));
3 nctx.softConstrPorts.add(new Tuple<BoolExpr, String>(
4     (ctx.mkEq(srcp, nctx.portMap.get("null")) ),"vpn_auto_conf"));
5 nctx.softConstrPorts.add(new Tuple<BoolExpr, String>(
6     (ctx.mkEq(dstp, nctx.portMap.get("null")) ),"vpn_auto_conf"));
```

To achieve the desired behavior of the framework, a positive weight should be assigned to these groups of constraints. This means that the values of `WPORT` and `WPROTOWILDCARD` should be changed from `-10` to `10`.
A specular reasoning can be applied to the soft constraints stating that the wildcard value should be assigned to the four components of the IP addresses of the placeholder rules:

```
1 nctx.softConstrWildcard.add(new Tuple<BoolExpr, String>(
2     ctx.mkNot(ctx.mkEq((IntExpr)nctx.ipFunctionsMap.get("ipAddr_1")
3     .apply(nctx.addressMap.get("wildcard")),srcAuto1)),
          "vpn_auto_conf"));
4 ...
```

The use of the `mkNot` Z3 boolean operator necessitates that the weight associated with this group of soft constraints be negative. Therefore, `WIPWILDCARD` values must be changed from `10` to `-10`. Alternatively, another way to correct this unintended behavior is to remove the `mkNot` operator and keep the current weight sign.

Finally, the last issue within the category of configuration errors was identified in the algorithm described in Listing 4.19. Specifically, the relevant section of the algorithm is as follows:

```
1  ...
2  int i=0;
3  for(BoolExpr boolProtect: protectExprs){
4      BoolExpr[] disProtect= new
           BoolExpr[disProtectExprs.size()-(i+1)];
5      for (int j=i+1; j<protectExprs.size();j++){
6          disProtect[z]=bl2[j];
7          z++;
8      }
9      z=0;
10     constraints.add(ctx.mkImplies(boolProtect,ctx.mkOr(disProtect)));
11     i++;
12 }
13 ...
```

For each element in the `protectExprs` list, an array of boolean expressions, `disProtect`, is created and populated with expressions indicating that successor nodes of the one applying protection must subsequently remove it. However, when the outer `for` loop iterates to the last element of the list, the inner `for` loop is completely skipped because `j` is equal to the size of `protectExprs`. As a result, a constraint is generated where the protection does not need to be removed. For example, if the node with IP address 50.0.0.2 is the last node in the `predecessors` list for a flow with `id=0`:

$$[\Rightarrow (\text{protect } |50.0.0.2|\, 0)(\text{or }\,)]$$

With this specific constraint, if the first condition is true (so the node 50.0.0.2 applies protection on traffic flow 0), no action is taken to remove the protection before the inspector or destination node, as intended. To address this issue, a condition has been added to ensure that the constraint is generated only if there is at least one node that removes the protection:

```
1  ...
2  if(disProtect.size>0)
3      constraints.add(ctx.mkImplies(boolProtect,ctx.mkOr(disProtect)));
4  ...
```

5.2.2 Redundancy Issue Resolution

The resolution of the redundancy issue affecting the framework's output required a significant rework of the VPNGateway function module. The algorithm responsible of the issue is the `generateSatisfiabilityConstraint()` method, and, in particular, the section that generates constraints about the configuration of the security associations:

```
1  ...
2  BoolExpr component2 = ctx.mkAnd(
3          nctx.equalIpAddressToVPNGwSA(flow.getCrossedTraffic(
4              ipAddress).getIPSrc(), srcConditions.get(srCount)),
5          nctx.equalIpAddressToVPNGwSA(flow.getCrossedTraffic(
6              ipAddress).getIPDst(), dstConditions.get(srCount)),
7          nctx.equalPort(flow.getCrossedTraffic(ipAddress).getpSrc(),
               portSConditions.get(srCount)),
8          nctx.equalPort(flow.getCrossedTraffic(ipAddress).getpDst(),
               portDConditions.get(srCount)),
9          nctx.equalLv4Proto(flow.getCrossedTraffic(
10             ipAddress).gettProto().ordinal(),
                   l4Conditions.get(srCount))
11         );
12 srCount++;
13 constraints.add(ctx.mkImplies(ctx.mkOr(opt1,opt2),component2) );
14 ...
```

In the current version, the `component2` boolean expression is constructed by concatenating all the fields of a single placeholder rule. These fields are retrieved from the maps defined in Listing 4.2, using the variable `srCount` as an index, which increments each time the `generateSatisfiabilityConstraint` method is called. Finally, a constraint is generated to imply that if the node is assigned an *ACCESS* or *EXIT* behavior, then `component2` must be satisfied. However, with this approach, if the VPN Gateway on the node is assigned an *ACCESS* or *EXIT* behavior, the constraints require that all placeholder rules be satisfied, resulting in the generation of a Security Association for each rule. This leads to redundant and unnecessary SAs in the output XML file.

To prevent generating a constraint for each placeholder rule, the rules can be aggregated into a list that includes all placeholder rules related to the specific flow. These rules can then be combined into a single boolean expression using the *OR* logical operator, and a constraint can be generated based on this aggregated expression:

```
1  List<BoolExpr> listSecondC = new ArrayList<>();
2  for(int i = 0; i < nRules; i++) {
3      BoolExpr singleComponent = ctx.mkAnd(
4              nctx.equalIpAddressToVPNGwSA(
5                  flow.getCrossedTraffic(ipAddress).getIPSrc(),
                       srcConditions.get(i)),
6              nctx.equalIpAddressToVPNGwSA(
7                  flow.getCrossedTraffic(ipAddress).getIPDst(),
                       dstConditions.get(i)),
8              nctx.equalPort(flow.getCrossedTraffic(ipAddress).getpSrc(),
                   portSConditions.get(i)),
```

```
9             nctx.equalPort(flow.getCrossedTraffic(ipAddress).getpDst(),
                 portDConditions.get(i)),
10            nctx.equalLv4Proto(flow.getCrossedTraffic(ipAddress)
11                .gettProto().ordinal(), l4Conditions.get(i)));
12    listSecondC.add(singleComponent);
13 }
14 BoolExpr[] tmp = new BoolExpr[listSecondC.size()];
15 BoolExpr component2 = ctx.mkOr(listSecondC.toArray(tmp)) ;
16 constraints.add(ctx.mkImplies(ctx.mkOr(opt1,opt2),component2) );
```

This way, just one placeholder rule is needed to be configured, and the final output of the framework does not show anymore redundant SAs. It is also coherent with the way the Security Associations are configured when the EndHost is equipped with VPN capabilities, as described in Listing 4.15.

5.2.3 New Optimization Strategies

As anticipated in subsection 5.1.3, the framework's functioning can be further optimized, significantly improving its performance.

The first enhancement in this regard involved re-evaluating the code implementation of the generation of constraints related to the protection requirments, based on the formulas (5.1), (5.2), and (5.3). While the current implementation adheres closely to the theoretical models, it does not fully consider the actual context in which it operates. Specifically, constraints related to the application and removal of protection on the traffic flows should be generated only for nodes that are capable of performing these actions, rather than indiscriminately generating constraints for all nodes, including those like forwarders and packet filters that lack this capability. To address this, a filter has been added to the various sections of the `createProtectionConstraints` method within the Checker module. This filter ensures that these constraints are generated only for nodes with a functional type of VPNGateway:

```
1  ...
2
3  // UNTRUSTED NODES HANDLING
4  for(AllocationNode predecessor: predecessors) {
5      if(predecessor.getTypeNF().equals(FunctionalTypes.VPN_GATEWAY)){
6          protectExprs.add((BoolExpr)nctx.protect.apply(
7              predecessor.getZ3Name(), ctx.mkInt(flow.getIdFlow())) );
8          disProtectExprs.add((BoolExpr)
               nctx.disProtect.apply(predecessor.getZ3Name(),
               ctx.mkInt(flow.getIdFlow())));
9      }
10 }
11 ...
12
```

```
13 // INSPECTOR / DESTINATION NODES HANDLING
14 for(AllocationNode predecessor: predecessors) {
15     if(predecessor.getTypeNF().equals(FunctionalTypes.VPN_GATEWAY)){
16         protectExprs.add((BoolExpr)nctx.protect.apply(
17             predecessor.getZ3Name(), ctx.mkInt(flow.getIdFlow())) );
18         disProtectExprs.add((BoolExpr)
               nctx.disProtect.apply(predecessor.getZ3Name(),
               ctx.mkInt(flow.getIdFlow())));
19     }
20 }
21 ...
22
23 // NODES WITH VPN CAPABILITIES HANDLING
24 if(predecessor.getTypeNF().equals(FunctionalTypes.VPN_GATEWAY)){
25     BoolExpr b1_1=ctx.mkEq(nctx.protect.apply(node.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkTrue() );
26     BoolExpr b1_2=ctx.mkEq(nctx.disProtect.apply(node.getZ3Name(),
          ctx.mkInt(flow.getIdFlow())), ctx.mkTrue() );
27     BoolExpr b1= ctx.mkOr(b1_1,b1_2);
28 }
29 ...
```

Another contribution of this thesis towards optimizing the framework has been the introduction of the set of constraints described in Listing 4.13 and Listing 4.14 within the VPNGateway module, which were originally present only in the EndHost class. With this addition, even fewer placeholder rules need to be configured, resulting in a lighter Z3 model and a clearer final output.

Finally, the optimization improvement that likely had the greatest impact on the framework's performance was the rework of the pruning function `minimizePlaceholderRules()`. The intended purpose of this method is to return the maximum number of placeholder rules that need to be defined and automatically configured by exploiting pruning strategies. However, the previous implementation was limited to simply counting the number of flows crossing the node and returning that number.

The new pruning strategy applied is fairly straightforward in its approach: the first filtering step considers only security requirements of the type *PROTECTION_PROPERTY*, marking these as candidates for potential pruning. The algorithm then avoids duplicate requirements by checking the identifier of each examined requirement. If the requirement is unique, it is added to the list of relevant properties. The method finally returns the size of this list of relevant properties, representing the minimized number of placeholder rules needed.

```
1 private int minizimePlaceholderRules() {
2     List<Flow> allProperties =
          source.getFlows().values().stream().collect(Collectors.toList());
3     List <Flow> interestedProperties = new ArrayList<>();
4     for(Flow p : allProperties) {
5         if(p.getRequirement().getOriginalProperty().getName().equals(
```

```
6             PName.PROTECTION_PROPERTY)) {
7             boolean toAdd=true;
8             for(Flow p2 : interestedProperties) {
9                 if(p.getRequirement().getIdRequirement() ==
                      p2.getRequirement().getIdRequirement())
10                toAdd = false;
11            }
12            if(toAdd)
13                interestedProperties.add(p);
14        }
15    }
16    int minimizedRules = interestedProperties.size();
17    return minimizedRules;
18 }
```

Listing 5.3: New implementation of pruning function

5.3 Performance Evaluation of the Enhanced Implementation

To evaluate the correctness of the fixes for the identified issues and the effectiveness of the new optimization strategies, a specific battery of tests has been developed. Each test case has been crafted to target specific aspects of the framework, such as flow path and constraint generation, placeholder rule pruning, and Security Association configuration, thereby facilitating a precise evaluation of the effects of the implemented modifications. The complexity of the network topology and the number of security requirements provided as input are varied according to the testing needs, whether that involves tracing the framework's execution through debugging or simply analyzing the final output in terms of correctness and performance.

5.3.1 Test Case 1

The Test Case 1 has been designed with the objective of analyzing the framework's behavior towards flow path generation and protection constraints on inspector and destination nodes. To reach the objective, the provided input must:

1. include multiple *WEBCLIENT* nodes to verify the correct generation of flow paths;
2. feature a node where a VPN Gateway will be allocated prior to the inspector or destination node, to validate the correct generation of protection constraints;
3. be sufficiently simple to facilitate detailed execution tracing through debugging;

Following these requirements, the crafted network topology is the following:

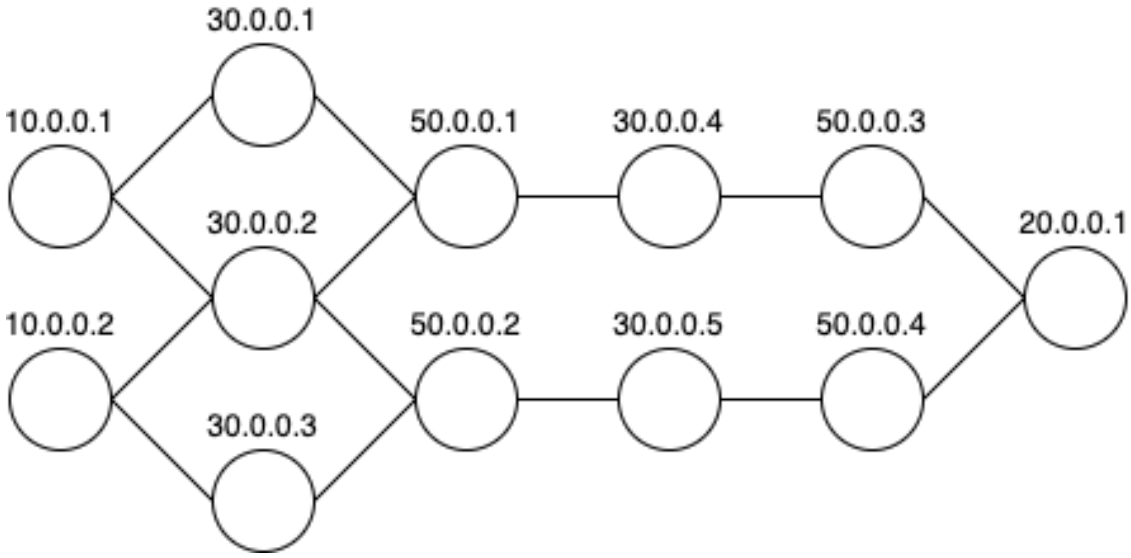
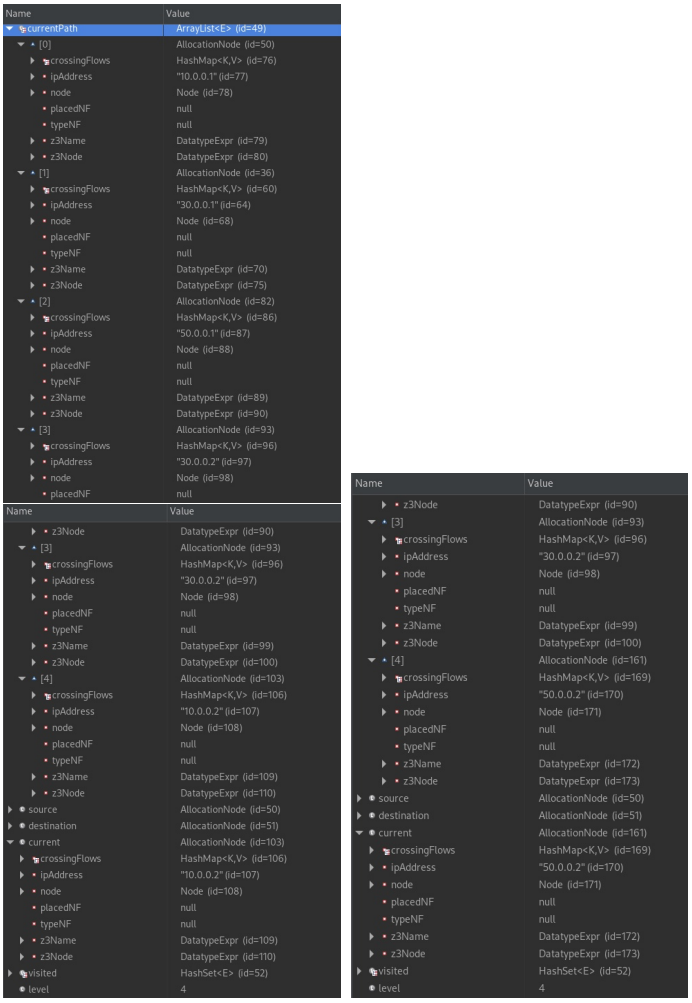


Figure 5.2: Test Case 1: Network Topology

The nodes 30.0.0.1, 30.0.0.2, and 30.0.0.3 enables the framework to attempt generating a flow path that, starting from one of the two *WEBCLIENTS*, also traverses the second. As each node can be visited only once during the recursive flow path generation, this topology represents the simplest configuration that allows for such an investigation.

The execution of the framework, provided with the input XML file corresponding to the Figure 5.2 topology, confirms the expected corrected flow path generation:



(a) (b)

Figure 5.3: Test Case 1: Flow Path Generation Corrected

The first figure demonstrates that the framework attempted to insert the node 10.0.0.2 into the current flow path at `level=4`; however, as it is a *WEBCLIENT* node, it was correctly removed from the list. The second figure then shows that the node 50.0.0.2 was selected as the next hop for the path.

The second aspect to be verified with this test case is the protection constraint generated in the Checker module for the inspector and destination nodes. With the fix discussed in Listing 5.2.1, the resulting constraint is as follows:

```
[(and (=> (protect |50.0.0.1| 0)
          (or (disProtect |50.0.0.2| 0) (disProtect |50.0.0.4| 0)))
      (=> (protect |50.0.0.2| 0)
          (or (disProtect |50.0.0.4| 0))))]
```

In the previous implementation, the last element of the constraint would have been `(=>(protect |50.0.0.4| 0)(or))`, which would imply that the node 50.0.0.4 could apply protection to traffic flow 0 without any subsequent node removing that protection. In contrast, the current implementation avoids generating this constraint entirely, addressing the problem directly.

5.3.2 Test Case 2

Test Case 2 was designed to verify whether the redundancy issue has been resolved with the implemented modifications concerning the configuration of Security Associations SAs within the VPNGateway module. As previously analyzed, the issue tends to arise when the framework is required to manage a network topology where multiple flows (and, consequently, multiple placeholder rules) are associated with a single Security Requirement. In a simple topology, such as the one illustrated in Figure 4.2, where only a single traffic flow is computed, the issue does not manifest. Consequently, an appropriate and efficient test input for this purpose should generate at least two traffic flows, while remaining as simple as possible.

The chosen network topology is the following:

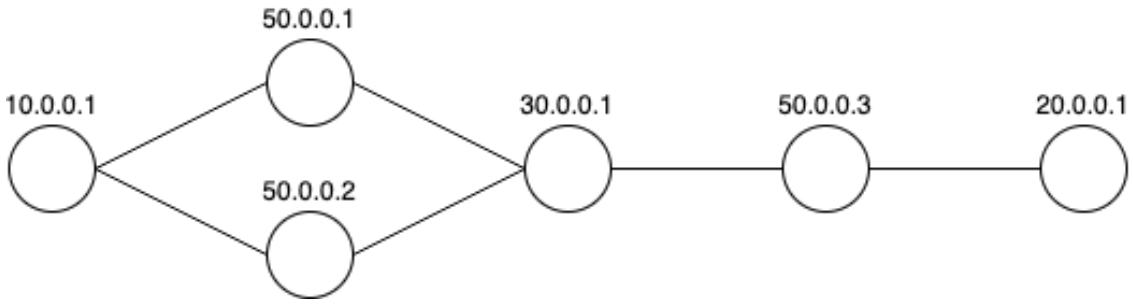


Figure 5.4: Test Case 2: Network Topology

In this topology, two flows are expected to be generated—one traversing the 50.0.0.1 node and the other the 50.0.0.2 node—with both crossing the 50.0.0.3 node. By specifying the node 30.0.0.1 as untrusted in the input file, the framework is forced to allocate a VPN Gateway with *ACCESS* behavior on both 50.0.0.1 and 50.0.0.2, and a VPN Gateway with *EXIT* behavior on 50.0.0.3. This way, the non-modified version of VEREFOO should generate two identical SAs on the node 50.0.0.3, while the newer implementation of the framework should overcome the redundancy issue.

With the theoretical expectations for the framework's computation outcome established, the next step is to provide the updated framework with the input XML file corresponding to the crafted network topology and observe the actual produced output, which is as follows:

```
1 <NFV xsi:noNamespaceSchemaLocation="./xsd/nfvSchema.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
2     <graphs>
3         <graph id="0">
4         <node name="10.0.0.1" functional_type="WEBCLIENT">
5             <neighbour name="50.0.0.1"/>
6             <neighbour name="50.0.0.2"/>
7         </node>
8         <node name="50.0.0.1" functional_type="VPNGateway">
9             <neighbour name="10.0.0.1"/>
```

```
10              <neighbour name="30.0.0.1"/>
11              <configuration name="AutoConf">
12                  <vpnGateway>
13                  <securityAssociation>
14                      <behavior>ACCESS</behavior>
15                      <startChannel></startChannel>
16                      <endChannel></endChannel>
17                      <source>10.0.0.1</source>
18                      <destination>20.0.0.1</destination>
19                      <protocol>ANY</protocol>
20                      <src_port>*</src_port>
21                      <dst_port>*</dst_port>
22                  </securityAssociation>
23                  </vpnGateway>
24          </configuration>
25          </node>
26          <node name="50.0.0.2" functional_type="VPNGateway">
27              <neighbour name="10.0.0.1"/>
28              <neighbour name="30.0.0.1"/>
29              <configuration name="AutoConf">
30                  <vpnGateway>
31                  <securityAssociation>
32                      <behavior>ACCESS</behavior>
33                      <startChannel></startChannel>
34                      <endChannel></endChannel>
35                      <source>10.0.0.1</source>
36                      <destination>20.0.0.1</destination>
37                      <protocol>ANY</protocol>
38                      <src_port>*</src_port>
39                      <dst_port>*</dst_port>
40                  </securityAssociation>
41                  </vpnGateway>
42              </configuration>
43          </node>
44          <node name="30.0.0.1" functional_type="FORWARDER">
45              <neighbour name="50.0.0.1"/>
46              <neighbour name="50.0.0.2"/>
47              <neighbour name="50.0.0.3"/>
48          </node>
49          <node name="50.0.0.3" functional_type="VPNGateway">
50              <neighbour name="20.0.0.1"/>
51              <neighbour name="30.0.0.1"/>
52              <configuration name="AutoConf">
53                  <vpnGateway>
54                  <securityAssociation>
55                      <behavior>EXIT</behavior>
56                      <startChannel></startChannel>
57                      <endChannel></endChannel>
```

```
58                  <source>10.0.0.1</source>
59                  <destination>20.0.0.1</destination>
60                  <protocol>ANY</protocol>
61                  <src_port>*</src_port>
62                  <dst_port>*</dst_port>
63              </securityAssociation>
64              </vpnGateway>
65          </configuration>
66      </node>
67      <node name="20.0.0.1" functional_type="WEBSERVER">
68          <neighbour name="50.0.0.3"/>
69      </node>
70      </graph>
71  </graphs>
72  <Constraints>
73      <NodeConstraints/>
74      <LinkConstraints/>
75  </Constraints>
76  <PropertyDefinition>
77      <Property name="ProtectionProperty" graph="0" src="10.0.0.1"
            dst="20.0.0.1" lv4proto="ANY" src_port="null"
            dst_port="null" isSat="true">
78          <protectionInfo>
79              <untrustedNode node="30.0.0.1"/>
80          </protectionInfo>
81      </Property>
82  </PropertyDefinition>
83  <profile name="hybrid"/>
84 </NFV>
```

As expected, despite the fact that node 50.0.0.3 is traversed by multiple flows, no redundant SAs appear in the final output. In contrast, with the previous implementation, a separate SA is present for each flow:

```
1   ...
2
3   <node name="50.0.0.3" functional_type="VPNGateway">
4           <neighbour name="20.0.0.1"/>
5           <neighbour name="30.0.0.1"/>
6           <configuration name="AutoConf">
7               <vpnGateway>
8               <securityAssociation>
9                   <behavior>EXIT</behavior>
10                  <startChannel></startChannel>
11                  <endChannel></endChannel>
12                  <source>10.0.0.1</source>
13                  <destination>20.0.0.1</destination>
```

```
14                  <protocol>ANY</protocol>
15                  <src_port>*</src_port>
16                  <dst_port>*</dst_port>
17              </securityAssociation>
18              <securityAssociation>
19                  <behavior>EXIT</behavior>
20                  <startChannel></startChannel>
21                  <endChannel></endChannel>
22                  <source>10.0.0.1</source>
23                  <destination>20.0.0.1</destination>
24                  <protocol>ANY</protocol>
25                  <src_port>*</src_port>
26                  <dst_port>*</dst_port>
27              </securityAssociation>
28              </vpnGateway>
29          </configuration>
30      </node>
31
32  ...
```

5.3.3 Test Case 3

The next aspect of VEREFOO that needs to be assessed is its performance and scalability. The previous test cases were intentionally designed to be as simple as possible to isolate the specific aspects or issues under analysis, facilitating easier evaluation. However, their low complexity also means they do not provide a reliable indication of the framework's true performance capabilities.

For this reason, Test Case 3 is designed solely to represent a more complex use case, with more nodes interacting. This setup provides a more reliable assessment of the framework's performance and scalability. the objective is to evaluate the performance impact of the modifications introduced to address the redundancy issue. Additionally, the effectiveness of the reworked pruning function will be measured by comparing the tool's performance both with and without the function applied.

The network topology illustrated in Figure 5.1 fits the requirements: with multiple clients and servers, and numerous flow paths to consider during the allocation and configuration problem resolution, it provides a comprehensive test of the framework's optimization capabilities and handling of complex scenarios.

The first version of the framework tested was the one with the original implementation of the `minimizePlaceholderRules` method. After twelve hours of computation on a workstation equipped with a 12-core Intel i7-1255U CPU and 40 GB of RAM, the framework had still not found a solution, indicating that the input size is beyond the current framework's processing capabilities.

However, when the new implementation of the pruning function was used, the framework was capable of producing a correct solution within eleven seconds of computation (10245 milliseconds), demonstrating the effectiveness of the approach.

Chapter 6

Conclusions

In this thesis, a detailed analysis of VEREFOO was conducted, which proved essential in identifying issues affecting the framework's implementation, understanding their nature, and determining solutions for addressing them. The framework was enhanced by introducing new optimization strategies aimed at improving its performance while maintaining the correctness of the output.

An in-depth research on the current landscape of network security configuration, along with the challenges and issues that arise when such tasks are performed manually, underscores the importance of a tool like VEREFOO, which is capable of automatically allocating and configuring security services. Furthermore, the literature review on the topic of automatic configuration of firewalls and VPNs highlighted specific areas where current research is lacking and demonstrated how VEREFOO could contribute to advancing the field.

With the foundational research phase complete, a high-level overview of the framework's functioning was provided, highlighting its key components and the most relevant modules for examination. The framework was dissected and carefully analyzed, with a primary focus on the algorithms responsible for generating constraints, on the Z3 model generated by the embedded solver and on the final output. This analysis uncovered several issues affecting the VEREFOO implementation, including errors in path and constraint generation, incorrect weight assignment to soft constraints and redundant Security Associations in the final output. The impact of these issues on the framework's correctness and output quality was thoroughly evaluated. Additionally, new optimization strategies were developed by recontextualizing the theoretical formulas within the actual VEREFOO environment and by completely reworking a pruning function that was not operating as intended. In particular, the generation of protection constraints has been restricted to network nodes capable of applying and removing protection on the traffic. This approach reduces the number of constraints fed to the solver for the automatic allocation and configuration problem, ultimately resulting in a lighter and cleaner Z3 model.

Finally, a battery of tests was developed to evaluate the effectiveness of both the solutions to the identified issues and the newly implemented enhancements. Each test case was crafted to target a specific aspect of the framework, with the design philosophy for each carefully explained.

In conclusion, the VEREFOO framework has proven to be a practical, automated solution for the allocation and configuration of VPNs within a network. Nonetheless, further work could be pursued to enhance its implementation and expand its capabilities.

In the future, heuristic studies could be conducted to achieve a meticulous balance between performance and optimality without compromising the correctness of the solution. Additionally, VEREFOO's ability to automatically configure VPNs could be integrated with its already well-developed capability for configuring packet filters. A final compelling extension would be the integration of an automatic reconfiguration feature, allowing the framework to respond dynamically when an attack on the network is detected, thus enabling a true fully automated and agile cybersecurity management.

Bibliography

[1] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, pp. 1559–1576, 2023. [Online]. Available: https://doi.org/10.1109/TDSC.2022.3160293

[2] ——, "Automated optimal firewall orchestration and configuration in virtualized networks," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110402

[3] D. Bringhenti and F. Valenza, "Greenshield: Optimizing firewall configuration for sustainable networks," *IEEE Transactions on Network and Service Management*, 2024.

[4] D. Bringhenti, R. Sisto, and F. Valenza, "A demonstration of VEREFOO: an automated framework for virtual firewall configuration," in *9th IEEE International Conference on Network Softwarization, NetSoft 2023, Madrid, Spain, June 19-23, 2023*, C. J. Bernardos, B. Martini, E. Rojas, F. L. Verdii, Z. Zhu, E. Oki, and H. Parzyjegla, Eds. IEEE, 2023, pp. 293–295. [Online]. Available: https://doi.org/10.1109/NetSoft57336.2023.10175442

[5] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: state of the art and research trends," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–37, 2023.

[6] M. Masdari, S. S. Nabavi, and V. Ahmadi, "An overview of virtual machine placement schemes in cloud computing," *J. Netw. Comput. Appl.*, vol. 66, pp. 106–127, 2016. [Online]. Available: https://doi.org/10.1016/j.jnca.2016.01.011

[7] F. Valenza, C. Basile, D. Canavese, and A. Lioy, "Classification and analysis of communication protection policy anomalies," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 2601–2614, 2017. [Online]. Available: https://doi.org/10.1109/TNET.2017.2708096

[8] E. Al-Shaer, H. H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE J. Sel. Areas Commun.*, vol. 23, no. 10, pp. 2069–2084, 2005. [Online]. Available: https://doi.org/10.1109/JSAC.2005.854119

[9] Splunk. (2024) State of security 2024: The race to harness ai. [Online]. Available: https://www.splunk.com/en_us/form/state-of-security.html

[10] V. Business. (2024) 2024 data breach investigations report. [Online]. Available: https://www.verizon.com/business/resources/reports/dbir/

[11] IBM. (2024) Cost of a data breach report 2024. [Online]. Available: https://www.ibm.com/reports/data-breach

[12] K. Hitomi, "Automation — its concept and a short history," *Technovation*, vol. 14, no. 2, pp. 121–128, 1994. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0166497294901015

[13] S. Ivanov, M. Kuyumdzhiev, and C. Webster, "Automation fears: Drivers and solutions," *Technology in Society*, vol. 63, p. 101431, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0160791X20300488

[14] W. K. Edwards, E. S. Poole, and J. Stoll, "Security automation considered harmful?" in *Proceedings of the 2007 Workshop on New Security Paradigms, White Mountain Hotel and Resort, New Hampshire, USA - September 18-21, 2007*, K. Beznosov and A. D. Keromytis, Eds. ACM, 2007, pp. 33–42. [Online]. Available: https://doi.org/10.1145/1600176.1600182

[15] P. Verma and A. Prakash, "FACE: A firewall analysis and configuration engine," in *2005 IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2005), 31 January - 4 February 2005, Trento, Italy*. IEEE Computer Society, 2005, pp. 74–81. [Online]. Available: https://doi.org/10.1109/SAINT.2005.28

[16] J. Wu, X. Chen, Y. Zhao, and J. Ni, "A flexible policy-based firewall management framework," in *International Conference on Cyberworlds 2008, Hangzhou, China, 22-24 September 2008, Proceedings*. IEEE Computer Society, 2008, pp. 192–194. [Online]. Available: https://doi.org/10.1109/CW.2008.134

[17] B. Zhang and E. Al-Shaer, "On synthesizing distributed firewall configurations considering risk, usability and cost constraints," in *7th International Conference on Network and Service Management, CNSM 2011, Paris, France, October 24-28, 2011*. IEEE, 2011, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/6103995/

[18] A. K. Bandara, A. C. Kakas, E. C. Lupu, and A. Russo, "Using argumentation logic for firewall configuration management," in *Integrated Network Management, IM 2009. 11th IFIP/IEEE International Symposium on Integrated Network Management, Hofstra University, Long Island, NY, USA, June 1-5, 2009*. IEEE, 2009, pp. 180–187. [Online]. Available: https://doi.org/10.1109/INM.2009.5188808

[19] P. Adão, C. Bozzato, G. D. Rossi, R. Focardi, and F. L. Luccio, "Mignis: A semantic based tool for firewall configuration," in *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society, 2014, pp. 351–365. [Online]. Available: https://doi.org/10.1109/CSF.2014.32

[20] A. El-Hassany, P. Tsankov, L. Vanbever, and M. T. Vechev, "Network-wide configuration synthesis," *CoRR*, vol. abs/1611.02537, 2016. [Online]. Available: http://arxiv.org/abs/1611.02537

[21] A. Saadaoui, N. B. Y. B. Souayeh, and A. Bouhoula, "Automated and optimized fdd-based method to fix firewall misconfigurations," in *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, D. R. Avresky and Y. Busnel, Eds. IEEE Computer Society, 2015, pp. 63–67. [Online]. Available: https://doi.org/10.1109/NCA.2015.31

[22] ——, "Formal approach for managing firewall misconfigurations," in *IEEE 8th International Conference on Research Challenges in Information Science,*

RCIS 2014, Marrakech, Morocco, May 28-30, 2014, M. Bajec, M. Collard, and R. Deneckère, Eds. IEEE, 2014, pp. 1–10. [Online]. Available: https://doi.org/10.1109/RCIS.2014.6861044

[23] A. En-Nouaary and M. Akiki, "An integrated framework for automated firewall testing and validation," in *Seventh International Conference on Information Technology: New Generations, ITNG 2010, Las Vegas, Nevada, USA, 12-14 April 2010*, S. Latifi, Ed. IEEE Computer Society, 2010, pp. 768–773. [Online]. Available: https://doi.org/10.1109/ITNG.2010.256

[24] E. Al-Shaer, A. El-Atawy, and T. Samak, "Automated pseudo-live testing of firewall configuration enforcement," *IEEE J. Sel. Areas Commun.*, vol. 27, no. 3, pp. 302–314, 2009. [Online]. Available: https://doi.org/10.1109/JSAC.2009.090406

[25] R. Isaacs, "Lightweight, dynamic and programmable virtual private networks," in *2000 IEEE Third Conference on Open Architectures and Network Programming. Proceedings (Cat. No.00EX401)*, 2000, pp. 3–12.

[26] Z. Fu and S. F. Wu, "Automatic generation of ipsec/vpn security policies in an intra-domain environment," in *Operations & Management, 12th International Workshop on Distributed Systems, DSOM 2001, Nancy, France, October 15-17, 2001. Proceedings*, O. Festor and A. Pras, Eds. INRIA, Rocquencourt, France, 2001, pp. 279–290. [Online]. Available: http://www.simpleweb.org/ifip/Conferences/DSOM/2001/DSOM2001/proceedings/S8-3.pdf

[27] Y. Yang, C. U. Martel, and S. F. Wu, "On building the minimum number of tunnels: an ordered-split approach to manage ipsec/vpn policies," in *Managing Next Generation Convergence Networks and Services, IEEE/IFIP Network Operations and Management Symposium, NOMS 2004, Seoul, Korea, 19-23 April 2004, Proceedings*. IEEE, 2004, pp. 277–290. [Online]. Available: https://doi.org/10.1109/NOMS.2004.1317665

[28] Y. Yang, Z. J. Fu, and S. F. Wu, "BANDS: an inter-domain internet security policy management system for ipsec/vpn," in *Integrated Network Management VII, Managing It All, IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003), March 24-28, 2003, Colorado Springs, USA*, ser. IFIP Conference Proceedings, G. S. Goldszmidt and J. Schönwälder, Eds., vol. 246. Kluwer, 2003, pp. 231–244.

[29] Y. Li and J. Mao, "Sdn-based access authentication and automatic configuration for ipsec," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 01, 2015, pp. 996–999.

[30] G. L. Millán, R. M. López, and F. Pereñíguez-Garcia, "Towards a standard sdn-based ipsec management framework," *Comput. Stand. Interfaces*, vol. 66, 2019. [Online]. Available: https://doi.org/10.1016/j.csi.2019.103357

[31] L. Firdaouss, A. Bahnasse, B. Manal, and Y. Ikrame, "Automated VPN configuration using devops," in *The 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2021) / The 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2021), Leuven, Belgium, November 1-4, 2021*, ser. Procedia Computer Science, N. Varandas, A. Yasar, H. Malik, and S. Galland, Eds., vol. 198. Elsevier, 2021, pp. 632–637. [Online]. Available: https://doi.org/10.1016/j.procs.2021.12.298

[32] P. B. Gentry, "What is a vpn?" *Inf. Secur. Tech. Rep.*, vol. 6, no. 1, pp. 15–22,

2001. [Online]. Available: https://doi.org/10.1016/S1363-4127(01)00103-0

[33] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[34] D. Bringhenti, R. Sisto, and F. Valenza, "Automating vpn configuration in computer networks," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2024.