
Politecnico di Torino

Master of Science in Computer Engineering

Master Degree Thesis

# Enhanced Deployment of Channel Protection Functions in Virtual Networks

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

**Candidate**
Angelo Floridia

Academic Year 2022-2023

# Summary

In today's world, automating Cybersecurity has become a crucial aspect of companies strategy defense to ensure security and reliability against constantly evolving network security threats. Through the exploitation of *Network Function Virtualization* (NFV) and *Software defined Networks* (SDN) it is possible to express a series of *Network Security Requirements* (NSRs) for any given network with the aim to use the paradigm of network virtualization to automate and optimize the allocation and the configuration of Network Security Functions (NSFs), such as packet filters and channel protection systems. Since a pure manual configuration is prone to human errors, it's been addressed an approach which allow network administrators to specify NSRs in a high-level language with a software capable to automatically translate them, establishing a graph of the network without policy conflicts and with NSFs allocated and configured automatically. These solutions are called *Refinement Tools* and are provided with *correctness-by-construction* verification approach.

When it comes to configure communication protected channels, it is mandatory to take into consideration not only the risks connected to a manual configuration, but also all the various variables present in the configuration of them. This is advised in order not to occur in low security configuration and/or data losses. To define these problems there have been studied taxonomies which classify these inconsistencies (i.e. Insecure communications, Unfeasible communication, Suboptimal walks, etc.).

The objective of this work is to create a *middleware*, capable of taking in input a correct and optimal VPN configuration and *translate* it to a real configuration for a IPSEC-based VPN software solution: Strongswan. This *Translator* is itself part of a bigger framework, VEREFOO, capable of translating NSR given in a high-level language by human to an automatic allocation and configuration of NSF in an *optimized, verified and correct* manner.

In VEREFOO, to model how traffic flows are forwarded and/or translated crossing the different nodes (Firewalls, NAT, VPN gateways) in the network have been proposed two approaches. The first takes into consideration the use of *Atomic Predicates*, each one identified by the IP quintuple, through which it is possible to calculate the set of minimal and totally disjunct set of predicates (atomic). The second one is dependent on the splitting of the traffic in the opposite way, which means creating fewer flows but combining them, having just one representative for all. There is no winner in these two approaches, but a series of pros and cons are discussed later in this document.

In this thesis, the focus was on the recognition of VPN tunnels in the virtualized network and the automatic generation of the respecting configuration files in the low-level language of Strongswan. To accomplish this goal, some tools have been used, like the *OpenSSH* project and the *SCP* protocol exploited through the Java Library *com.jcraft.jsch*. The network topology was analyzed, and its configuration has been sanitized and translated to be compatible with Strongswan and the new configuration files were sent to Virtual Machines, used as test environment.

Due to the non-readyness of the network configuration coming in output from VEREFOO for a real configuration implementation, some assumptions have been necessary, and some limitations were pointed out, but overall the tests have showed promising results, easily improvable in the future, confirming this way is respectable to follow.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

## 1.1 Introduction and Motivation

In today's world, automating Cybersecurity has become a crucial aspect of companies strategy defense to ensure security and reliability against constantly evolving network security threats. It is possible to express a series of *Network Security Requirements* (NSRs) for any given network with the aim to use the paradigm of network virtualization to automate and optimize the allocation and the configuration of Network Security Functions (NSFs), such as packet filters. The goal is to allow network administrators to specify NSRs in a high-level language and automatically translate them, establishing a graph of the network without policy conflicts and with NSFs allocated and configured automatically. These solutions are called *Refinement Tools* and are provided with *correctness-by-construction* verification approach.

Some Internet Service Providers (ISPs) have tried to create services to include security in their offers, with the end-user being just a client who would not worry about the safety of its infrastructure, but it is not an easy task, given that each ISP could have potentially hundreds of thousands of clients. Some studies led to the elaboration of a framework to enable the ISPs to provide *Security as a Service* (SECaaS) [1] to their clients, using *Network Function Virtualization* (NFV) and *Software defined Networks* (SDN).

The *NFV* allows using virtual machines and software processes instead of hardware and physical machines, controlled by an *hypervisor*, providing flexibility in terms of network function implementation;

The *SDN* allows centralizing the control power inside the single components of the network itself, securing the detachment of the *data plane* and the *control plane*, with the latter being fine-grained programmable.

But even *SDN* can be vulnerable if configurations are manually calculated. In fact, some studies in IoT [2] and Smart Homes [3], where this paradigm is heavily exploited, have been conducted on how to *automatically verify* and *allocate* the SDN configurations based on security policies given by network and security administrators and proved some limitations are still present.

As a matter of fact, virtualized networks are rising in presence nowadays and more ways of configuring Virtual Network Functions (VNFs) and its specification are in development. For this in Europe it was founded the ETSI (*European Telecommunications Standards Institute*) which aims to standardize the architecture. Despite that, there are a numerous different implementation of VNF and this led to the developing of some tool to help developers and administrators to use an high-level language to specify network policy and requirements [4].

However, current literature lacks solutions for packet transformers and models to forecast the network's behavior, which is necessary to optimize security functions allocation and configuration. It is challenging to model modern networks due to their stateful and diverse transformations.

Therefore, this thesis aims to propose and compare network modelling approaches to automatically allocate and configure security mechanisms in virtualized networks, considering high-level user requirements and perform an automatic deployment in a real virtualization environment.

## 1.2 Thesis description

In this section is briefly illustrated the remainder of this thesis:

- **Chapter 2**: describes Virtual Private Networks, how are they defined and which types of private connections exists, the technology behind them and the anomalies that a manual configuration can introduce;
- **Chapter 3**: provide a description of the framework used as main base and for which the work of this thesis aims to add new functionality, its model base, and the two main approach used to create the input for the SMT solver, the Atomic Flows and the Maximal Flows;
- **Chapter 4**: represent the introduction to the framework used to fulfill the thesis objective. It describes the background work that has already been done in traffic modelling and traffic description and the actual state-of-the-art in refinement of some NSF, like firewall and CPS;
- **Chapter 5**: contains the description of the thesis objectives;
- **Chapter 6**: describes the core contribution to this thesis, which goal is to deploy in a real virtualized environment the configuration of a Channel Protection System through the use of an Open Source software, Strongswan and an algorithm to translate and create a configuration file for it;
- **Chapter 7**: address the validation and testing of the algorithm of translation defined in the previous chapter, using a Virtualization software;
- **Chapter 8**: contains the conclusions, summarizing the overall approach to the translation algorithm and presents the improvements proposed for a future work;

- **Appendix A**: contains all the configuration steps to reproduce the environment and the tests, as it was a big part of this work;
- **Appendix B**: collection of XML schemas of test cases;

# Chapter 2

# Virtual Private Network

When it comes to secure online communication, Virtual Private Networks (VPNs) offer a powerful solution to safeguard information and traffic enciphering and authenticating. VPNs allow users on a host to connect to other hosts through a virtual tunnel, which aims at securing the communications. This technology provides Layer 3 end to end protection for homogeneous networks, like IP networks, regardless of the underlying network infrastructure.

## 2.1 Type of VPN

The VPN can be classified by *deployment* scenarios. Understanding the distinction between these VPN types is essential in determining the most suitable solution for specific requirements.

### 2.1.1 Site-to-Site

A site-to-site VPN enables secure communication between two or more geographically separate locations, such as in a corporate infrastructure, where there could be more branches of the company dislocated in several cities and/or countries. This scenario emulates the interconnection of them as they would be in one big Local Area Network, communicating each other using IP secure communication protocols like *IPsec* and *TLS* as they were physically interconnected.

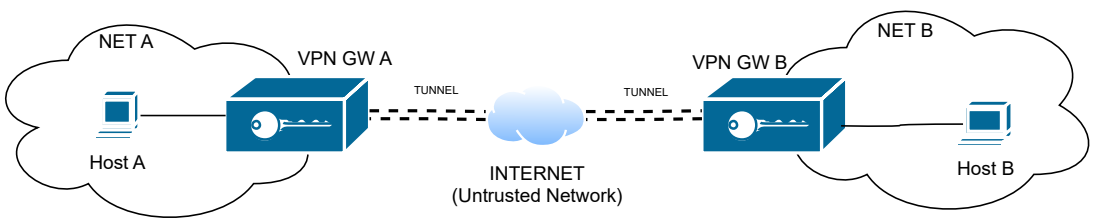


Figure 2.1. VPN model: Site-to-Site

As we can see in figure 2.1 the *Host A* in *Network A* can communicate securely with *Host B* in *Network B* through the use of a **tunnel**, instantiated by the two border gateways *VPN GW A* and *VPN GW B*, exchanging information through an insecure means like *internet*.

### 2.1.2 End-to-End

The end-to-end VPN focuses on securing communication between individual hosts or devices. The end hosts themselves have VPN capabilities, it means they act as a VPN gateway of their own and instantiate a tunnel regardless of their physical locations.

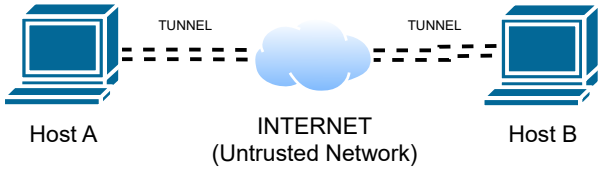


Figure 2.2. VPN model: End-to-End

As showed in figure 2.2, *Host A* and *Host B* are interconnected using a **tunnel** over an insecure means. This type of VPN is mostly used and suggested in case the data protection is crucial such as remote collaboration, accessing sensitive resources from external networks and so on. The encrypted and authenticated data at the source can only be read at the destination.

### 2.1.3 Remote-Connection

Also called *Access VPN*, it allows the secure connection of an individual user to a private network from a remote location. This kind of scenario is widely used nowadays by remote workers, which must connect to the corporate network to be able to securely access data, hosts, files and other kind of resources.

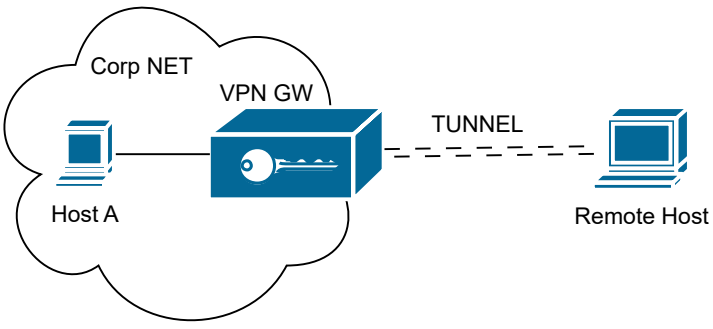


Figure 2.3. VPN model: Remote-connection

The figure 2.3 describe the scenario, with the **tunnel** instantiated between the *Remote Host* and the border VPN Gateway. This type of VPN ensures a remote user a secure gateway to connect to, in order to access the internal resources while maintaining the security of the organization's network.

## 2.2 VPN Technologies

VPN *tunnels* can be established at layer 2 as well as at lever 3 of the ISO/OSI stack. Here follows an introduction to both technologies, even though this work focuses only on the layer 3 VPN, since it is the only one technology that can be addressed by Software Defined Network paradigm. Among the popular VPN technologies, IPsec and TLS stand out as widely adopted protocols for establishing secure connections.

### 2.2.1 IPsec

The **Internet Protocol SECurity** is an integrated network protocol suite which allows having secure communications between networks and/or hosts by means of *encryption, authentication, encapsulation* as well as *protection* against *replay* attack over an untrusted network.

It provides several tools to establish *mutual authentication* and for the creation of *cryptographic keys* to authenticate the packets.

It is the technology used to achieve the goal of this thesis.

#### IPsec Security Architecture

The security architecture is given by some protocols included in the *IPsec* suite:

- *Authentication Header (**AH**)*: provide the possibility to authenticate the **whole packet** and to guarantee the integrity of it, in order to remove the possibility to *tamper* with the content of the packet (replay attacks). It is important to remind that there is **no encryption**, so the packet can be sniffed and read without keys, authorization or whatsoever. The integrity property is reached by using an *hash function* and a shared *secret* key.

- *Encapsulating Security Payload (**ESP**)*: this header provides authenticity, data integrity and confidentiality. While AH authenticates the whole packet, ESP authenticate **and** encrypt the **payload** of the packet, for data privacy. If used in *tunnel* mode, by the way, a new packet header is added and the ESP protection is extended to the whole inner packet.

- *Security Association (**SA**)*: is where the negotiated algorithm and keys to be shared for the AH or the ESP protocols are established. There are many different ways to build security associations like *pre-shared keys*, IKE and IKEv2 (*Internet key Exchange*), *Kerberos*, etc.

#### IPsec Modes

Finally, IPsec protocols can be used in two different modes:

- *Transport Mode*: in transport mode only the payload (actual data) of the IP packet is encrypted and/or authenticated while leaving the header untouched. This does not affect the routing, which is based on the non-modified header of the packet.
- *Tunnel Mode*: in this case, the all IP packet is secured by authentication and encryption. The secured packet is encapsulated in a larger IP packet with a new header used for the routing, adding an extra layer of security. This mode is mainly used for creating VPN tunnels.

### 2.2.2 TLS

The Secure Socket Layer, called today **Transport Layer Security**, allows using a tunnel for the entire traffic of a network communication or to secure an individual communication. It is used mainly when IPsec has some difficulties, for example in traversing NAT, firewalls and routers, since its original header encapsulation. TLS services can be accessed by web browsers through HTTPS protocol.

TLS VPNs leverage cryptographic protocols and certificates to establish secure connections between clients and servers. This technology ensures that data transmitted between the client and the VPN server remains confidential and protected from eavesdropping and tampering. TLS VPNs are often used for remote-access scenarios, allowing users to securely connect to corporate networks or access sensitive resources over the internet.

It acts at *presentation* layer, while IPsec acts at *network* layer (the 6th of the ISO/OSI stack), which means it is critical in case of a Denial of Service kind of attacks because the traffic has to be processed till the transport layer (4th), instead of being dropped at network layer (3rd).

## 2.3 VPN anomalies

Enforcing computer system security is a complex task requiring specific skills. Security administrators often lack on tool support to validate enforced policy compliance. *Communication Protection Policies* (CPPs), used to define communication protected channels, are difficult to manage and often sensitive, as mistakes can lead to data loss or security breaches. In [5] a set of nineteen policy anomalies have been addressed and analyzed. A model was introduced and, through First Order Logic formulas, has been possible the detection of the anomalies and the resulting strategies to avoid them.

Two taxonomies were instituted, classifying the nineteen anomalies. One based on the side effects and one, more technical and information-centered, based on where the anomalies arise.

The first one of the two taxonomy presented is being reported here, called *effect-based taxonomy*:

- **Insecure communications**: inadequacy, monitorability, skewed channels, asymmetric channels;
- **Unfeasible communications**: non-enforceability, out of place, filtered, L2;
- **Potential errors**: shadowing, exception, correlation, affinity, contradiction;
- **Suboptimal implementation**: redundancy, inclusion, superfluous, internal loop;
- **Suboptimal walks**: alternative path, cyclic path;

After the classification and after having built and tested the model, the researchers pose their attention to answering two main questions:

1. If the anomalies classified in the paper were really introduced in CPPs by network administrators;
2. If a more expert network administrator will be less keen to introduce anomalies compared to a less expert one;

To answer the two questions, they ran an experiment with 30 participants.

The results were very interesting: in fact, 93% of network administrator did introduce at least one anomaly and all nineteen anomalies were introduced by at least one administrator. This answers the first question, the purpose of the paper to model and search for anomalies on the network is indeed useful.

Moreover, from the experiment data they deduced that expert administrators introduce fewer anomalies, except for the *sub-optimal implementation* macro category: they tend to add more redundancy and superfluous anomalies in order to create a deeper defense strategy.

The evidence of the presence of the anomalies is not to think only in a manual configuration case scenario, but also in the automatic one. In fact, the VEREFOO framework, as described in the next chapter (3) allows the automatic configuration and allocation of Security Functions but after having executed an analysis of the configuration provided.

Finally, the work of this paper allows detecting incompatibilities, redundancies and errors introduced in communication policies based on human errors. This is also the purpose of this thesis work, introducing the translator algorithm to create secure communication tunnels and automatically deploy them, as described in chapter 6.

# Chapter 3

# VEREFOO framework

In the context of security automation, there have been recent proposals for implementing an automated approach to policy-based network security management systems. This includes leveraging technologies like *Network Functions Virtualization* (NFV) and *Software-Defined Networking* (SDN) to enhance network management and increase networking flexibility. Manual configuration of security functions is prone to errors and can be time-consuming, especially as networks become more complex. It is evident that producing a correct and optimized configuration manually is challenging. Automated approaches offer the potential to compute configurations more efficiently and eliminate human errors. Many of these approaches rely on formal methods to ensure solution correctness. Some tools even provide optimization features to minimize resource usage in addition to correctness. This chapter introduces the VEREFOO approach, which is based on formal methods and utilizes a *partial weighted* Maximum Satisfiability Module Theories (maxSMT) problem. By solving this problem, VEREFOO generates a *formally correct* configuration for the given Network Security Requirements while minimizing the number of firewalls and configured firewall rules. The chapter provides an overview of the tool's general structure and modules and presents theoretical concepts such as the distinction between Service Graph and Allocation Graph, along with the relevant constraints defined for the MaxSMT problem.

## 3.1 Introduction

To fulfill this goal, a novel approach has been developed and studied by means of Network Function Virtualization and the Software Defined Network paradigm. The result of some studies like [6] and [7] lies in helping not to introduce any errors in manual configuration and, moreover, to optimize and to verify them and finally, help the administrator to choose from a pool of security functions and propose candidate technology based on the network project.

VEREFOO (*VErified Refinement and Optimization Orchestrator*) [8] is a framework capable of translating *Network Security Requirements* (NSR) given in a high-level language by human being to an automatic allocation and configuration of *Network Security Functions* (NSFs) on a Service Graph, being them optimized, verified and correct.

To perform this role, the framework is based on the resolution of a *Maximum Satisfiability Modulo Theories* (MaxSMT) problem through the use of open-source Microsoft's *z3Opt* engine.

Based on its three pillars *Optimation, Optimality* and *Formal Correctness,* it has a dual objective:

- The optimal *Allocation* Scheme for NSFs;
- The optimal *Configuration* of the NSFs;

## 3.2 Model description

The model of VEREFOO, depicted in figure 3.1, is based on two kinds of graph, which can be given as input and from which the framework can generate the optimal solutions, and other components as follows:

- **NSRs set**: a group of *Network Security Requirements* needed to match with and satisfy the *Security Constraints*. They can be expressed by means of an High-Level or a Medium-Level language, depending on the level of expertise of the network administrator creating them;
- **Service Graph**: it is a logical topology created by network functions put together to form a complete end-to-end service, in which are allowed loops and multiple paths to reach destinations. It does not contain any *security requirements*, hence there will not be any *firewalls, VPN gateways* or *antispam filters*, etc. which will be added later in the Allocation Graph. An example of Service Graph is depicted in figure 3.2;

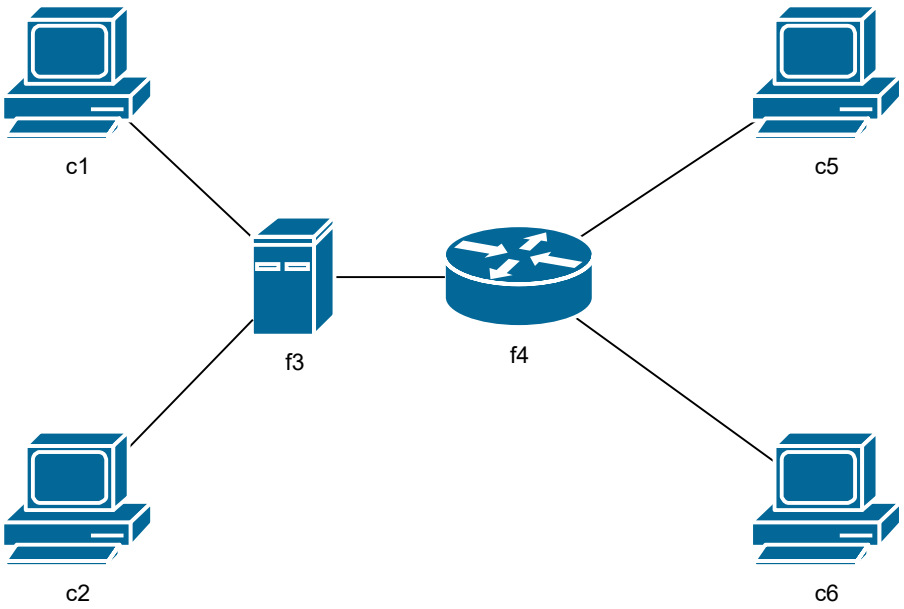


Figure 3.2. Service Graph

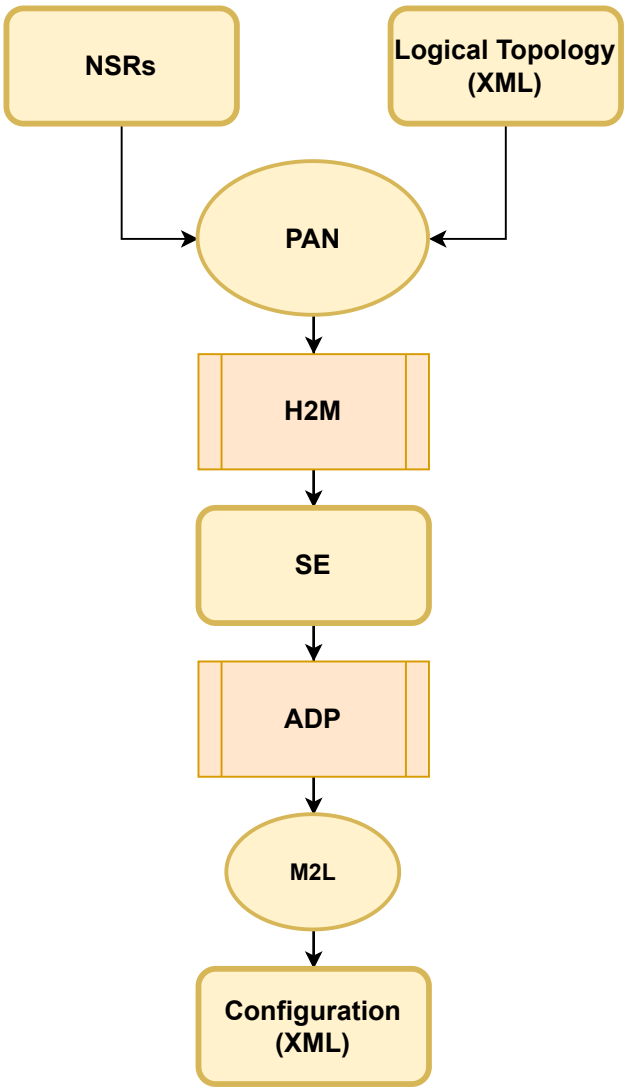


Figure 3.1. VEREFOO flow diagram

- **Allocation Graph**: it is a logical topology of network functions to form a complete end-to-end service, but, this time, between each node there will be another additional node called *Allocation Place* in which Network Security Functions could be allocated. The framework allocates security functions in the APs in an optimal way, using the remaining ones to allocate forwarders. The Allocation Graph (AG) is a logical topology that can be generated either from scratch or automatically generated based on the Service Graph. If an Atomic Predicate (AP) remains unused but is part of the path of at least one input requirement, a forwarder would take its place, forwarding each received packet.

  In VEREFOO, the process of automatically generating an *Allocation Graph* (AG) from a *Service Graph* (SG) involves adding a new AP on every link between two nodes. However, the service designer can impose constraints on the generation process. This includes forcing the allocation of a Network Security Function (NSF) on a specific AP or prohibiting the placement of a

new AP in a particular location where no network function can be placed. An example of Allocation Graph is depicted in figure 3.3;

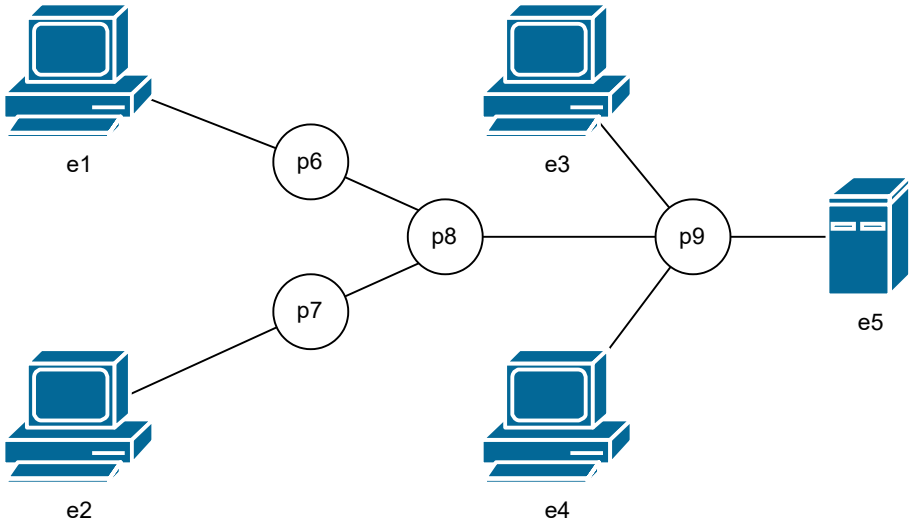


Figure 3.3. Allocation Graph

- **PAN module**: the *Policy ANalysis* module is responsible to perform a *conflict's analysis* between the Network Security Requirements given in input and detecting errors. A *non-enforceability report* is generated if conflicts cannot be resolved, otherwise it will determine the minimum set of requirements that need to be satisfied.

- **H2M module**: this module has the purpose to translate the *Security Requirements* given in input by the network administrator in high-level language to medium-level language security requirements for the framework, containing all the useful information to create the *Network Security Function* and automatically allocate them on the allocation graph;

- **SE module**: the *Network Function SElection* is in charge of respecting and satisfy the Network Security Requirements by choosing the right Network Security Functions from a pool of predefined ones (NF pool);

- **ADP module**: the core function is carried out by the *Allocation, Distribution and Placement* module. Its inputs consist of a group of network security requirements expressed in medium-level language, a list of conflict-free Network Security Functions, and either a Service Graph or an Allocation Graph. The ADP module then produces a new Service Graph that includes any newly allocated network security functions, in addition to those already present in the input graph, or a physical graph. Additionally, the module generates configuration for each allocated network security function thanks to a M2L module, which translates configurations from medium to low level; to do so it uses the z3 problem solver. The NSRs are introduced as **hard** constraints that must always be satisfied by the solver. On the other hand, other specifications and optimizations are introduced as weighted and optional **soft** constraints.

- **M2L module**: the last module is the *Medium-to-Low* (M2L) module, which takes the list of medium-level policy rules produced by the solver as input and translates them into a low-level language based on the actual implementation of the network functions.

### 3.2.1 Network Model

The description of the **network model** is achieved by the use of graphs. The representations of all possible network functions (i.e. firewalls, routers, VPN Gateways, etc.) is left to the nodes of the aforementioned graphs while the network functions connections are represented by the edges. In VEREFOO, the graph used is of the type *directed* which means each edge represent an unidirectional communication and, to have it both ways, we would need two edges.

In the model of a network, each node possesses specific properties that define the element allocated to that position. These properties include a set of input and output ports, which determine whether a packet with a specific header can traverse through that port. This is useful when it is needed to understand if packets are *allowed* to pass through that node or the passing is *denied*. Additionally, an essential characteristic is the *Transformation* function, denoted as $\tau$. Within a node, packets may undergo various transformations, such as header rewriting, encapsulation, de-encapsulation and label switching. Upon entering a node, a packet is examined against the input domains of the transformation function $\tau$, and the corresponding function is then applied to the packet.

Since this thesis focuses on Channel Protection Systems, it is important to categorize nodes based on the feature they could have in relation to the channels. There are 3 different node's category:

- **Untrusted Nodes**: refers to those nodes that must adhere to security requirements. Often the need for these kinds of nodes are overlooked by a network administrator that, by default, assumes that all nodes between a source and a destination are trusted. This latest approach is recommended to be avoided, except for the case in which the administrator has no knowledge at all of the topology of the network. Conversely, considering a set of untrusted nodes expands the solution space and often leads to resource savings. If not differently specified, it is safer to consider any nodes as untrusted;

- **Inspector Nodes**: refers to those nodes that must pass without any protection for the purpose of analyzing the content. Network security functions, such as Intrusion Detection Systems (IDS), typically require access to unencrypted network traffic for analysis and detection of malicious content. If this kind of nodes would not be present in the network, some conflicts may arise because the IDS can not read the payload of the packets, triggering unnecessary alarms or dropping useful traffic. However, even if enhancing the interoperability between security functions, employing inspector nodes may introduce additional delays due to the potential need for multiple encryption/decryption processes.

- **Trusted Nodes**: refers to those nodes where the decision to enforce security requirements is at the discretion of the solver. To illustrate this, we can denote $A$ as the set of all nodes, $U$ as the set of untrusted nodes, and $I$ as the set of inspector nodes. The set of trusted nodes, denoted as $T$, can be obtained by subtracting the sets of untrusted and inspector nodes from the set of all nodes:

  $$T = A - U - I$$

  In other words, any nodes that do not belong to the sets of untrusted or inspector nodes will be automatically categorized as trusted by the solver. This implies that the network administrator is not required to explicitly specify the nodes falling into this category.

## 3.3 Atomic Flows vs Maximal Flows

To formally define the traffic originated at a node and forwarded in the network is a delicate task, composed by modeling packets, paths, transformation and, more in general, traffic flows between each node. Let us remember that this models must guarantee the formally corrected representation of the traffic and, at the same time, an efficient computational algorithm. In [9] two model approaches are introduced, called *Atomic Flows* and *Maximal Flows*, to overcome the drawbacks the already existent tools carry, challenging efficiency and scalability. The two approaches, needed to represent optimized traffic flows, have been then implemented in the VEREFOO framework, to solve the reachability and refinement problem performing an optimal allocation and configuration of NSFs.

### 3.3.1 Background

A packet is represented by a **predicate** that is computed based on specific fields, primarily the header. Packets sharing the same values in these fields belong to the same class and are associated with the same predicate. Consequently, all nodes encountered in the network treat these packets consistently. It is important to note that a predicate represents the network traffic of a packet. To determine the forwarding domains and the transformation behavior for a packet, the predicate it belongs to is exclusively considered. Therefore, it is essential to represent the rules in the nodes' ACL, the rules defined in the forwarding tables, and the domains of the transformation function using predicates. This ensures that the same model used for network traffic is applied. By doing so, the predicate describing incoming packets can be compared with the predicates characterizing each encountered node. This facilitates decision-making regarding forwarding and transformation behavior based on matches with the node's rules or domains.

The ability to compare two predicates is a crucial aspect to consider when selecting a model for their representation. Specifically, the chosen predicate model should support various comparison operations such as *intersection, union* and *negation*. The ultimate objective is to fully represent the entire network, including all its components, as distinct sets of predicates that comprehensively describe its forwarding and transformation behavior.

For clarity, it is possible to address a predicate as an IP packet and the sub-predicates are its fields (IP source, IP destination, port source, port destination, protocol). The real predicate representing the IP packets is the conjunction of all the sub-predicates which can represent a single value, several values in a range or, with the wildcard symbol "*" a full range of values.

**Traffic Flows** are utilized to describe the collection of predicates and nodes (a flow) crossing a network, denoted with $F$. Each traffic flow represents the behavior of a particular packet class along a path, encompassing the packet's exit from the source node, its forwarding through intermediate nodes, and any transformations it undergoes while traversing from source to destination. The purpose of utilizing traffic flows is to comprehensively describe the behavior of an entire network solely through the set of traffic flows ($F$). Specifically, when aiming to model network traffic for refining security properties, the focus is on a subset of possible flows known as *interesting flows*. These flows are selected based on specific sources and destinations defined by security policies. The interesting flows are determined by processing the given Network Security Requirements and the configuration of encountered nodes. Selecting an appropriate flow model is crucial to efficiently describe network behavior, particularly in computing how an entering packet is forwarded and transformed while traversing various nodes (e.g., NAT, load balancers, VPN gateways, firewalls).

Two distinct and alternative models for describing traffic flows have been studied, implemented, and compared in a previous work [9] and it follows a presentation and comparison of them.

### 3.3.2 Atomic Flows

Based on the idea presented in [10] on the *Network Reachability Problem*, an **atomic predicate** is a set of simple and minimal predicates coming from the partition of complex predicates (i.e. representative of NAT input or source classes, etc.).

From having a collection of network predicates, the set of completely disjoint, minimal and fully representative predicates (referred as *Atomic*) can be calculated in such way that each predicate from the initial set can be expressed as a combination of a subset of the predicates from the second set.

**Definition 3.3.1.** *Given the set $\mathcal{P}$ of predicates, the derived set of Atomic Predicates $\{p_1, ..., p_k\}$ is such if satisfies five properties:*

1. $p_i \neq false,\ \forall i \in \{1, ..., k\}$.
2. $\bigvee_{i=1}^{k} p_i = true$
3. $p_i \wedge p_j = false,\ if\ i \neq j$
4. each predicate $P \in \mathcal{P},\ P \neq\ false$, is equal to the disjunction of a subset of atomic predicates $P = \bigcup_{i \in \mathcal{S}(P)} p_i$, where $\mathcal{S}(P) \subseteq \{p_1, ..., p_k\}$
5. $k$ is the *minimum* number such that the set $\{p_1, ..., p_k\}$ satisfies the above four properties

From a specific predicate, it is feasible to compute the set of corresponding *Atomic Predicates* as follows:

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

Given two sets of Atomic Predicates $P_1 = \{b_1, ..., b_l\}$ and $P_2 = \{d_1, ..., d_m\}$, the set of Atomic Predicates corresponding to their union $P_3 = A(P_1 \cup P_2) = \{a_1, ..., b_k\}$ is equal to:

$$ai = b_{i_1} \wedge d_{i_2} | ai \ \neq \ false, \ i_1 \in \{1, ..., l\}, \ i2 \in \{1, ..., m\} \tag{3.2}$$

From this set of atomic predicates, it is possible to derive the set of the correspondent Atomic Flows.

**Algorithm 1** Computing Atomic Predicates

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

```
for i ← 1, n do
    compute A({P_i}) using 3.1
end for
for i ← 2, n do
    compute A({P_1, ..., P_i}) from A({P_1, ..., P_{i-1}}) and A({P_i}) using 3.2
end for
```

The **Atomic Flow** approach employs APs to describe the traffic traversing the network and configure firewalls and other network functions with rules expressed solely using atomic predicates. In general, it describes the network and its behavior using predicates from the AP set. The approach begins by identifying the *interesting* predicates and then computing the corresponding set of *Atomic Predicates* as previously described. The *interesting* predicates are those associated with nodes related to the given *Network Security Requirements* (NSR). This includes predicates representing the traffic generated from the source node (source traffic), traffic reaching the destination node (destination traffic), input traffic classes and transformation behavior for encountered transformers along the paths.

Once the set of Atomic Predicates for all the interesting predicates in the network is computed, the next step involves generating all possible Atomic Flows for each user requirements. These Atomic Flows are then used as input for the MaxSMT solver to allocate and configure the necessary Network Security Functions.

**Definition 3.3.2.** *A flow* $f = [n_s, t_{sa}, n_a, ..., n_h, t_{hi}, n_i, ..., n_k, t_{kd}, n_d]$ *is defined* ***atomic*** *if each traffic* $t_{ij} \in \mathcal{B}$*, where* $\mathcal{B}$ *is the set of Atomic Predicates computed from the set of interesting predicates.*

One significant advantage of this approach is that the predicates are inherently unique, allowing each predicate to be associated with a distinct integer identifier.

Consequently, the solver can operate using simple integers to represent the traffic, eliminating the need for more complex representations like the previously mentioned Predicate class. This streamlined approach significantly improves the solver's resolution performance, as it enables a more efficient problem model. Additionally, since many operations involve intersections and unions, working with sets of integers is less complex compared to sets of more intricate predicate representations.

It is important to note that while the Atomic Flows approach may not produce the smallest possible number of configured rules in an absolute sense, it achieves the smallest number of disjointed rules, which is not necessarily the absolute optimal solution. Another drawback of this approach is the initial computational time required to generate the set of Atomic Predicates from the set of interesting predicates. This step is computationally intensive as it involves processing one interesting predicate at a time and computing its intersection with all other APs in the set before adding it, ensuring that the set only contains disjointed traffics.

### 3.3.3 Maximal Flows

This approach is based on achieving the maximum integration of the flows, going in the exact opposite direction as with the atomic flows. In fact, rather than with the atomic approach where the aim was to maximize the number of *simple* flows, here the aim is to minimize the number of flows, but equally representative of the network flows, aggregating them in **maximal flows**.

A **Maximal Flow** [11] is a subset of all flows, which only contains flows that are not in other subset of flows and behave consistently when crossing the various nodes of the network. Basically, the goal is to aggregate all the flows with the same behavior under a specific subset, reducing the number of representative flows in the network. In this case, the traffic flows are modeled as a list of alternating nodes and predicates. However, instead of using atomic predicates, the employed predicates express the disjunction of multiple IP quintuples.

**Definition 3.3.3.** *Called $F_r$ the set of all possible flows of the network, the corresponding set of Maximal Flows $\mathrm{F}_r^M$ matches the following definition:*

$$\mathrm{F}_r^M = \{\mathrm{f}_r^M \in F_r \mid \nexists f \in F_r.(f \neq \mathrm{f}_r^M \wedge \mathrm{f}_r^M \subseteq f)\}$$

The flows that behave in the same way are aggregated into the same Maximal Flow, ensuring consistent treatment by the network. The design and resolution of the MaxSMT problem are then modeled using the set $\mathrm{F}_r^M$, which has the same expressiveness as $F_r$ but a smaller size. One major advantage of this approach is the significantly faster computation of the set $\mathrm{F}_r^M$ compared to the set of Atomic Flows. Unlike the previous approach, the algorithm for $\mathrm{F}_r^M$ computation does not require initial computation time for traffic flows. However, a drawback is that the traffics exchanged between nodes within all Maximal Flows are not disjoint and unique. Therefore, they cannot be associated with integer identifiers like Atomic Predicates, but complex data structure must be used (i.e. BDDs, Wildcard Expressions). The solver used by VEREFOO, specifically works with 13 fields: 4 integers for the source IP address, 4 integers for the destination IP address, 2 integers for the source port

range, 2 integers for the destination port range and a string for the protocol type. The increased number of variables provided to the solver has a *significant impact* on the overall resolution performance.

### 3.3.4 Comparison

Atomic Flows and Maximal Flows are two contrasting approaches used to model network traffic behavior. As per the results obtained in [9], it is feasible to say that the Atomic Flows approach focuses on maximizing the number of simple flows by employing Atomic Predicates. These predicates are unique and can be associated with integer identifiers, resulting in a streamlined problem representation. This approach offers improved resolution performance, as operations involving intersections and unions are less complex when performed on sets of integers. However, generating the set of Atomic Predicates from the interesting predicates requires significant initial computational time.

On the other hand, Maximal Flows aim to minimize the number of flows while maintaining representativeness. Flows with similar behavior are aggregated into Maximal Flows, reducing the overall number of representative flows. The computation of Maximal Flows is faster compared to Atomic Flows, as it doesn't require initial computation time. However, the traffics within Maximal Flows are not disjoint and unique, necessitating the use of complex data structures for their representation. The increased number of variables used in Maximal Flows can impact the resolution performance. Both approaches offer distinct trade-offs in terms of flow representation, computation time, and resolution performance, catering to different requirements in modeling network behavior.

Here follows a recap about advantages and disadvantages of both the approaches, as depicted in table /refFlowComparison:

Atomic flows:

+ : simpler identifier (i.e. one Integer)

− : number of flows is greater than Maximal flow approach

− : time needed for the initial computation of the flows.

Maximal flows:

+ : number of flows is considerably less than atomic flows.

+ : time needed for the initial computation of the flows is less, plus it is a *recursive, parallelizable* algorithm.

− : Complex identifier; i.e 4 Integer for IPv4 source, 4 for IPv4 destination, 2 for source port, 2 for destination port, 1 String for protocol type;

The approaches have been tested for the resolution of two of the major security problem presented in this thesis: the refinement problem and the verification

Table 3.1. Atomic Flow and Maximal Flow comparison

| Type | Identifier | #flows | Initial computation time |
|---|---|---|---|
| Atomic | + | − | − |
| Maximal | − | + | + |

problem of requirements and policies, on which are based the allocation of security functions. The *refinement* problem addresses the translation from High-level language to a Low-level language policies to transform them to system configurations. The *verification* problem instead aims to guarantee that a security function is correctly enforced in a system, using an SMT solver (MaxSMT).

Based on the tests that have been conducted in [9], to solve the first problem regarding *verification* and *reachability*, the *Maximal Flow* approach proves to reach better performances, mostly due to the initial computation time for the Atomic flows approach, other than the fact that the latter generates much more flows.

On the contrary, in solving the problem of *refinement* it is the *Atomic Flow* approach which performs better, since the SMT problem calculation phase weights much more in respect to initial calculation phase, and it is faster with simpler Atomic Flow's identifier with respect to the Maximal Flow complex one's.

# Chapter 4

# Network Security Functions in VEREFOO

## 4.1 Introduction

NFV paradigm exploits the representation in software of network and security functions, the Virtual Network Functions (VNFs). These can be deployed in virtual environments like Virtual Machines or Containers hosted in high computing powered servers. Network Security Functions are a subset of VNFs, which exploit SDN to create software instances.

## 4.2 Network Security Requirements

As seen in chapter 3.2, one of the input of the VEREFOO framework is expressed by Network Security Requirements, which formalize the conditions for the configuration of the Virtual Network Functions.

At the time of writing this thesis, VEREFOO contains different possible requirements:

- **Reachability Requirements**: gives VEREFOO information about source nodes being able to contact and connect to destination nodes. The framework has to make sure *NO* packets will be dropped in the path by the use of packet filters (firewall, see 4.3.1), configuring them in whitelist (all communication blocked, except the specified one) or blacklist (all communication allowed, except the specified one);

- **Isolation Requirements**: gives VEREFOO information about source nodes *NOT* being able to contact and connect to destination nodes. The framework has to make sure all the packets will be dropped in the path by the use of packet filters (firewall, see 4.3.1), configuring them in whitelist (all communication blocked, except the specified one) or blacklist (all communication allowed, except the specified one);

- **Communication Protection Requirements**: gives VEREFOO information about the establishment of communication protection between two nodes, based on a set of policies (CPS, see 4.3.2). Since this work is mainly focused on this, it is further elaborated in the next section;

### 4.2.1 Communication Protection Requirement

The Communication Protection Requirements are used and provided if the objective is to ensure a protected communication channel between two end-nodes, which can have different feature as authentication, secrecy, etc. These kinds of requirements are defined by means of properties and protection information:

- **RuleType**: represents the security requirement that should be satisfied, in the context of Secure Communications requirements it assumes the value *"ProtectionProperty"*.
- **FilteringCondition**: define the criteria for selecting the traffic that needs to be protected. These conditions include:
  - *IPSrc*: the source IP address of the traffic flow that requires protection.
  - *IPDst*: the destination IP address of the traffic flow that requires protection.
  - *portSrc*: the source port at the transport level of the traffic flow that requires protection.
  - *portDst*: the destination port at the transport level of the traffic flow that requires protection.
  - *transportProto*: the transport-level protocol of the traffic flow that requires protection.

  To represent the IP addresses (*IPSrc* and *IPDst*), the conventional dot-decimal notation is used, which represents the IP address as a series of four numbers separated by dots.

  $$ip1.ip2.ip3.ip4$$

  where $ip_i$, $\forall i \in \{1,2,3,4\}$ can be represented as integers in the range from [`0-255`] (inclusive) or as wildcard elements denoted by "*". The wildcard symbol allows for a unified representation of both network addresses and their corresponding netmasks. For example, the representation `10.0.0.*` can be used to express the endpoint present in the network `10.0.0.0/24` while `192.168.*.*` characterizes the network `192.168.0.0/24`.

  The source and destination transport-level ports, *portSrc* and *portDst*, can be specified as single numbers or as intervals of numbers within the range of [`0 - 65535`]. For example, if it is required to block traffic flow between the source `10.0.0.2` and the destination `20.0.0.5` when the source port numbers fall within the interval [`1, 4460`], the packets characterized by a source

port number within this interval would be blocked to satisfy the isolation requirement.

Lastly, he *transportProto* element in the Network Security Requirements represents the layer-4 protocol used above the IP layer. It can take values such as [`TCP, UDP, ICMP`], or the wildcard "*". When the wildcard is used, it indicates that the property must be satisfied while considering the possibility that the source can send packets using TCP, UDP or ICMP protocols.

- **ProtectionInfo**: contains information about the topology and the technology used:
  - *Security Technology*: specifies the technology to be adopted in order to enforce the security requirement. TLS/IPsec, ref 2.2;
  - *Authentication Algorithm*: Algorithms employed for packet authentication and integrity and all the cryptographic algorithm;
  - *Encryption Algorithm*: Algorithms used for the encryption (if necessary) of the packets, ref 2.2
- **TopologyInfo**:
  - *Untrusted/Inspector Nodes*: nodes in which the packets must flow encrypted (untrusted) or unencrypted (inspector, i.e. IDS). Each node of the path is considered an untrusted node by default;
  - *Untrusted Links*: links in which the packets must flow encrypted, which is the default configuration. If a link should be considered as *trusted*, it must be defined as such.

Being $\mathcal{N}$ the set of all network nodes and $\mathcal{F}$ the set of all network flows, two predicates are required for the formulation of security protection requirements:

$$PROTECT\ (n, f) \Rightarrow Boolean\ with\ n \in \mathcal{N}, f \in \mathcal{F} \tag{4.1}$$

$$UNPROTECT\ (n, f) \Rightarrow Boolean\ with\ n \in \mathcal{N}, f \in \mathcal{F} \tag{4.2}$$

To effectively fulfill the creation of the Channel Protection Requirements, the following conditions must be met:

1. For each traffic flow within the requirement, when encountering an *Untrusted Node*, the number of predecessor nodes responsible for adding a protection layer (PROTECT) must exceed the number of nodes involved in removing it (UNPROTECT). This condition is vital to ensure the security of the traffic passing through the untrusted node.

2. For each traffic flow falling under the requirement, when encountering an *Inspector Node* or the destination node, the number of predecessor nodes responsible for adding a protection layer (PROTECT) must match the number of nodes removing it (UNPROTECT). This condition is vital to ensure that traffic remains unsecured while passing through the *Inspector Node* or arriving at the destination node.

3. For each traffic flow falling under the requirement, if there exists an *Untrusted Link*, the number of predecessor nodes responsible for adding a protection layer (PROTECT) must exceed the number of nodes removing it (UNPROTECT). This condition is vital in safeguarding the security of traffic when traversing the untrusted link.

Listing 4.1 is an example of how the Protection Requirement can be transposed in the XML input file.

Listing 4.1. XML Schema of a Protection Requirements

```
<PropertyDefinition>
    <Property graph="0" name="ProtectionProperty" src="10.0.0.1"
    dst="30.0.5.2" dst_port="80">
        <protectionInfo encryptionAlgorithm="AES_128_CBC"
            authenticationAlgorithm="SHA2_256">
            <untrustedNode node="20.0.0.3"/>
                <securityTechnology>TLS</securityTechnology>
                <securityTechnology>IPSEC</securityTechnology>
        </protectionInfo>
    </Property>
</PropertyDefinition>
```

## 4.3 Network Security Functions

This chapter introduces two of the most discussed network technology in the context of the framework contemplated in this thesis. The firewall, as packet filters, and its configuration and the Channel Protection System.

### 4.3.1 Firewall

A *Firewall* is a Network Function which monitors and manages incoming and outgoing packet traffic based on Security Policy defined manually or automatically. The establishment of policies based on the type of network and the implementation of security rules to permit or restrict access, are key factors in preventing potential hacker or malware attacks.To ensure maximum protection against cyber attacks, it is crucial to correctly configure the firewalls using a Filtering Policy. Manual configuration of the firewall settings can be time-consuming, prone to human errors, and may result in conflicts or anomalies.

To overcome the limitations associated with manual configuration, the concept of Security Automation is utilized. Security Automation employs automated control processes to define security feature policies, effectively minimizing the occurrence of human errors and enhancing overall security.

#### Related work

In the context of VEREFOO, firewalls and packet filters have been the Security Function most developed and tested. There are a numerous amount of papers

about this subject, like [11, 12]. Furthermore, it has been studied the reaction of the aforementioned Firewalls in relation to network structure modification and update [13], pursuing a correctness-by-construction approach, limiting the number of operations and achieve the optimality by use of a MaxSMT solver. Different virtualization environment has been tested, like Virtual Machines and Kubernetes [14].

### Conflict Analysis

The use of the SDN paradigm introduces some edge cases, considered as anomalies, which must be addressed and managed. For example, preventing forwarding errors or the non-reachability of sections of a network.

In order to avoid human distractions in configuration and the consequential trial and errors stages, which brings to a non-complete vision of the real network behavior, automated software tools based on Formal Verification approaches has been developed, able to model the network and preventing the creation of anomalies (or minimizing it [15]), improving the creation of verified and optimal policies, solving a MaxSMT problem with wide coverage of NSF and with high performance [16].

The manual configuration of the Filtering Policy is associated with certain challenges and risks, including:

- High probability of configuration errors: Without an automatic validation mechanism, the likelihood of introducing errors or redundancies in the rules is significant. There is no built-in check to ensure the correctness of the rules or to identify conflicts with existing rules on the devices.
- Complexity and potential for failure with multiple Firewalls: Managing multiple Firewalls within a network increases complexity and the probability of failures. Each Firewall requires its own set of filtering policies, resulting in a fragmented rule repository that is harder to maintain and coordinate.
- Impact on internal network security: Incorrect configurations can have detrimental effects on the internal security of the network. Misconfigurations may inadvertently allow unauthorized access or compromise the integrity of the network, posing risks to sensitive data and resources.

To mitigate these challenges and risks, the adoption of Security Automation approaches can help streamline the configuration process, ensure rule consistency, and enhance the overall security of the network.

The conflicts can occur when rules are misconfigured within a single policy, resulting in **intrapolicy conflicts**, or they can arise between policies on different devices, leading to **interpolicy conflicts**. These misconfigurations, whether within a policy or between policies, can undermine the effectiveness of the security measures implemented in the system.

The first type of conflicts, the **Intrapolicy Conflict** happens when rules present in a single security devices makes frictions and can be categorized as follows:

- **Intrapolicy Shadowing**: when a rule is preceded by another rule with a different action, it becomes obscured and will never be applied;
- **Intrapolicy Correlation**: correlation conflicts arise when two rules are related to each other but have different types of actions. This dependency can lead to ambiguity in defining the security policy;
- **Intrapolicy Exception**: a rule acts as an exception to the following rule if it has different actions and the subsequent rule matches a superset of the rule itself;
- **Intrapolicy Redundancy**: redundancy occurs when a rule is overshadowed by another rule, rendering it unused as packets will never match it.

The second type of conflicts, the **Interpolicy Conflict**, happens between rules on different devices. Conflicts occur when a *downstream* device permits traffic that is blocked by one of the *upstream* devices, or when an *upstream* device permits traffic that is blocked by one of the *downstream* devices. Conversely, all intermediate devices should allow or exclude any traffic that is allowed or protected by both the upstream and downstream devices, enabling the flow to reach its destination.

They can be categorized as follows:

- **Interpolicy Shadowing**: unlike intrapolicy shadowing, this conflict occurs between two rules of different devices. It arises when an upstream policy blocks a portion of the traffic that is allowed by a downstream policy;
- **Interpolicy Spuriousness**: traffic is considered spurious when the upstream device allows certain traffic that is actually blocked by the downstream device. This situation has significant implications for network security, since it permits unwanted traffic to pass through the network.

### 4.3.2 Channel Protection System

*Channel Protection Systems* refer to security measures and technologies implemented to safeguard communication channels and ensure the confidentiality, integrity, and availability of data transmitted over those channels. These systems aim to protect the channels through which information is exchanged between two or more entities, such as individuals, organizations, or devices.

Data transmission is one of the most critical feature of network security and can be achieved by means of Secure Channels. It is possible to think of a channel like a tunnel, starting at the source node - which may or may not add a protection layer - and ending at the destination node - which may o may not remove the protection layer. This kind of communication technology is able to enforce secure communications between network entities, satisfy the security requirements given by a network administrator and assure data integrity, data confidentiality and authentication.

The manual configuration of secure channels is prone to human error which can, potentially, causes a big breach in the security texture of the systems itself. Also, the networks are becoming more and more complex and taking everything in mind while manually configure security is an hard challenge as shown in 2.3.

### Related Work

Many studies have been conducted on the automatic generation of security policies and on the verification of the security requirements manually written [17], on optimizing the number of tunnels looking at the global correctness [18] and avoiding conflicts among policies [19, 20].

Moreover, given the fact that cryptographic tunnels has to be implemented in at least two nodes of the network (start and end points), it is not simple as seen with firewalls to implement, especially if large number of nodes are present [21, 22].

Another very important criteria to take into consideration discussing CPSs is the fact that not only automatic configuration of the CPSs must be achieved, but also the automatic *allocation* of them. In this, none of the above has succeeded, mostly because they are based on physical topology and not on a virtualized environment.

### Channel Protection System model

Channel Protection System can be modeled as Virtual Private Network tunnels, as seen in chapter 2. This security property is applied and removed on the packets at the start node and at the end node of a tunnel. Since the manual configuration of this kind of secure channel is trivial, some automatic configuration tool has been engineered, as proposed in [23].

Each VPN Gateway is configured with a *Security Policy*, which expresses the configuration about protection adding and removing, cryptographic algorithms, etc. The policy can be manually configured or can be automatically computed by the ADP Module of VEREFOO.

A policy is marked by:

- the *action* $\Omega$;
- the *policy rules* $\Psi$.

In the context of VEREFOO, the schema of the policy rules is modeled as follows:

- **Behavior**: describe the action taken for each packet. It can assume the following values:
  - *ACCESS*: a VPN Gateway *adds* an external header to the traffic received if it needs to be protected from this stage and if it matches the security policies of the Gateway. The external header has the VPN Gateway IP address as source and the terminal VPN Gateway IP address as destination;
  - *EXIT*: a VPN Gateway *removes* an external header from the traffic received if it needs to be without protection from this stage and if the packet matches the security policies of the Gateway;

  - *FORWARD*: a VPN Gateway simply *dispatches* the traffic received to the next node without adding or removing headers;

- **Start Channel**: it is the starting node of the tunnel;
- **End Channel**: it is the ending node of the tunnel;
- **Conditions**:
  - *source*: IP address of the source node of the traffic;
  - *destination*: IP address of the destination node of the traffic;
  - *source port*: port of the source node of the traffic;
  - *destination port*: port of the destination node of the traffic;
  - *protocol*: protocol used to generate the traffic;
- Authentication Algorithm: set of algorithm to guarantee the authentication and the integrity of the packets flowing through the tunnel;
- Encryption algorithm: set of algorithm to guarantee the encryption of the packets flowing through the tunnel;

In order to better understand the *Behavior* of a VPN Gateway, it is possible to use the concepts of *PROTECT* and *UNPROTECT* seen in chapter 4.2.1.

If exist $g$ a node with VPN Gateway functionality and $f$ a flow traversing the Gateway $g$:

- The **ACCESS** behavior can be expressed by:

$$PROTECT(g, f) \wedge \neg(\ UNPROTECT(g, f)\ ) \tag{4.3}$$

- The **EXIT** behavior can be expressed by:

$$\neg(\ PROTECT(g, f)\ ) \wedge UNPROTECT(g, f) \tag{4.4}$$

- The **FORWARD** behavior can be expressed by:

$$\neg(\ PROTECT(g, f)\ ) \wedge \neg(\ UNPROTECT(g, f)\ ) \tag{4.5}$$

It is worth noting that a VPN Gateway is capable of selecting only one of the three available behaviors for a given pair (node, flow):

$$(ACCESS) \wedge (EXIT) \wedge (FORWARD) \tag{4.6}$$

Thanks to nowadays hardware and computational power, even the end nodes can act as a VPN Gateway themselves, creating an end-to-end tunnel with another node. This approach is sometimes preferred thanks to the resource consumption decrease on the VPN Gateway *central* nodes.

### Conflict Analysis

Channel Protection Systems implementation is based on a list of Security Requirements to fulfill, which express the required protection. This process, called *refinement*, dwell in selecting the right security technology and enforce the protection policy at the network nodes, optimizing if possible (like aggregate individual channels in a single tunnel). Unfortunately, it can introduce anomalies like incorrect path, poor security, and so on, leading to a cost raise for the infrastructure.

In [23] is presented an approach to address the problem of anomalies correctness, conflict detection and resolution based on *penalties* and *backtracking* algorithms.

In [24] is presented an innovative classification of Protection Policy irregularities and a formal model capable of identifying anomalies in implementations at various network level, using FOL (First Order Logic) axioms to guarantee accuracy and performances.

Another approach based on efficient algorithms is presented in [25], exploiting Binary Decision Diagrams (BDD's).

# Chapter 5

# Thesis Objectives

The past chapters have showed the VPN technology and its peculiarity followed by an introduction to the state of the art of virtual networking, introducing the VEREFOO framework.

Based on this and what has been said in the introduction about the vulnerability and disruption any human error can introduce when configuring large networks, this work aims at improving and enhance the automation of network management tools, in order to always make it safer and limiting human mistakes.

The paragraphs that follow use the aforementioned technologies, together with the relative related works, to bring the automation of creating security configuration in VPN scenarios to the next level, covering what is now done manually by network administrators.

In particular, the VEREFOO framework, as already discussed, is able to elaborate and produce a graph representative of the network, with all the nodes configured for their functions.

In the context of VPN Gateways, this work focuses on bringing the configuration contained in this graph to an actual configuration of a software capable to instantiate a real VPN tunnel. Not only there will be the **translation** from the middle level language of VEREFOO to a low level one, used by the final software, but also it will be done in an automatic approach.

The software that is going to be used for this work is **Strongswan**, chosen for its full IPsec support and for its license. It features a robust and highly regarded open-source VPN solution that provides secure communication across networks. It provides advanced features and support for various encryption protocols and extensive configuration options, making it suitable for a wide range of deployment scenarios, ensuring secure and encrypted connectivity for both organizations and individuals.

The main objectives of this work can be summarized as follows:

1. The first objective of the work is to create a **new approach**, developed in the form of an algorithm, that can effectively recognize the network topology and, based on the number and on the types of nodes to be able to call the right procedure to begin the process of **translation**. The translation process

is specifically tailored to handle various VPN scenarios, including site-to-site, end-to-end and remote-access. The detail of this process is detailed in chapter 6, providing a comprehensive understanding of the methodology employed.

2. Another important objective of this thesis was to focus on automating the entire process of **generating** new configuration files and seamlessly deploying them into Virtual Machines in order to **automatically establish** new secure tunnels between the machines. The Strongswan's configuration file *swanctl.conf* was first studied in depth in its documentations in order to understand its details, followed by a theoretical validation of test cases as detailed in chapter 7, providing valuable insights into the efficacy and reliability of the implemented solution.

3. The last objective of this thesis was to write a comprehensive appendix that serves as a guide for future developers and students. This appendix offers step-by-step **instructions** on navigating the software solution introduced in this work, providing a detailed walkthrough of the virtual environment's construction, which is based on Linux Virtual Machines. By documenting the configuration and setup procedures, this appendix aims to empower individuals to replicate and extend the software solution with ease, fostering a collaborative and knowledge-sharing environment.

Lastly, a series of rigorous tests were conducted to validate the efficacy and performance of the newly developed algorithms. These tests served as a means to assess the extent to which the set goals were accomplished through the utilization of the innovative Translator Java class and the integration of Virtual Machines. The outcomes of these tests provided compelling evidence of the successful implementation and functionality of the proposed solution. Detailed analysis and results of these tests are presented in the subsequent chapters, shedding light on the effectiveness and reliability of the developed system.

# Chapter 6

# Approach for the StrongSwan Configurations Translation

## 6.1 Background

In this chapter, the use of the upon earlier introduced VPN (4) is exploited to automate the deployment of this technology on real case scenario VNFs.

As seen in 3, the VEREFOO framework receives in input some NSR and a Logical Topology and return in output a complete topology graph with configurations in `XML` format, i.e. Virtual Private Network tunnels configurations.

Here it is described the *Translation Algorithm* developed to perform the translation of a VPN configuration from the `XML` format to a configuration for a software, *StrongSwan*.

## 6.2 Strongswan Introduction

*Strongswan* [26] is an *Open-source*, modular and portable *IPsec-based VPN solution*, implementing the *Internet Key Exchange (IKE)* protocol that allows securing IP traffic in policy and route-based *IPsec* scenarios from simple to very complex.

Server side, it supports Linux, Android, FreeBSD, macOS, iOS and Windows. Most devices and Operating Systems today natively support the IKEv2 protocol, and this is the reason why StrongSwan was chosen.

StrongSwan is distributed under the *GPLv2 license*, which means it is open source and allows us to use all the features for free. Besides, we can also participate on the developing of the software itself and further improve it.

Since the work in [27], where the tool was used for the first time in the context of this framework, Strongswan was updated to a new version which made the old *TranslatorStrongSwan class* of VEREFOO deprecated.

## 6.3 Translation Algorithm

The Translation Algorithm is a crucial component in the process of converting VPN Gateway configuration to Strongswan's configuration. This algorithm takes the allocation graph (described in chapter 3.2) as its input and generates the necessary configuration files as output. By using the information provided by the allocation graph, the Translation Algorithm ensures a seamless transition between the two different VPN gateway configurations.

The primary objective of the Translation Algorithm is to accurately map the network functions and their corresponding configurations from VPN Gateway to Strongswan. This includes translating security policies, authentication algorithms, encryption algorithms, and other relevant parameters. By executing the Translation Algorithm, the required configuration files are generated, enabling the successful migration of VPN Gateway configuration to Strongswan's configuration.

The Translation Algorithm plays a vital role in simplifying the migration process and reducing manual efforts. It ensures that the converted configuration aligns with Strongswan's requirements and guarantees a secure and efficient VPN infrastructure.

Unfortunately, this comes with some limitations, because the starting *schema* contains an high-level abstraction of the network and contains too few information about the real topology. This and other limitation will be discussed later in chapter 7.5.

### 6.3.1 Network Topology

In order to develop the translation algorithm, a comprehensive understanding of the data extractable from the XML file is essential. The analysis of the XML file, as depicted in Listing 6.1, reveals a notable limitation: the inability to define subnet masks for the networks. Consequently, all examples and tests have been conducted using a default /24 subnet mask. This limitation is acknowledged and discussed in Chapter 7.5, along with potential areas for future improvement.

An important assumption that has been made is to **ignore** the **EXIT** behavior in the VPN schema and make it symmetrical, identical to the **ACCESS** one. In fact, it is very common that a tunnel is symmetric, if the traffic need protection going out from the gateway (*ACCESS behavior*), then also the one coming in input (*EXIT behavior*) for that destination will be protected.

By considering these factors and incorporating relevant data from the XML file, the translation algorithm can be designed and implemented effectively. This algorithm will play a crucial role in facilitating the conversion process of VPN configurations, ensuring consistency and preserving the intended security measures.

### 6.3.2 The Translator

Here is portrayed the algorithm used to delineate what kind of connection exists between the nodes of the virtual network generated.

Listing 6.1. VPN Gateway declaration in Allocation Schema XML file

```
<node functional_type="VPNGATEWAY" name="192.168.57.3">
  <neighbour name="10.0.0.4"/>
  <neighbour name="192.168.57.4"/>
  <configuration description="vpn_gw">
    <vpngateway>
      <vpnName>moon</vpnName>
      <securityAssociation>
        <behavior>ACCESS</behavior>
        <startChannel>192.168.57.3</startChannel>
        <endChannel>192.168.57.4</endChannel>
        <source>10.0.0.-1</source>
        <destination>10.0.1.-1</destination>
        <protocol>TCP</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
        <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
        <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
      </securityAssociation>

      <securityAssociation>
        <behavior>EXIT</behavior>
        <startChannel>192.168.57.4</startChannel>
        <endChannel>192.168.57.3</endChannel>
        <source>10.0.1.-1</source>
        <destination>10.0.0.-1</destination>
        <protocol>TCP</protocol>
        <src_port>*</src_port>
        <dst_port>*</dst_port>
        <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
        <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
      </securityAssociation>
    </vpngateway>
  </configuration>
</node>
```

The *pseudo-code* of the algorithm is represented in Algorithm 2.

Here follows a description of the variable expressing the algorithm:

- *nVpnGW*: the number of VPN Gateways present in the output graph. For the algorithm to start, this value must be greater than `0` which means the presence at least of one gateway;
- *siteToSiteNodes*: nodes involved in site to site tunnels. It contains all the nodes which have at least one security association in which the *source* network of the traffic is different from the *startChannel* `AND` the *destination* network is different from the *endChannel*;
- *endToEndNodes*: nodes involved in end to end tunnel. It contains all the nodes which have at least one security association in which the *source* network of the traffic is equal to the *startChannel* `AND` the *destination* network is equal to the *endChannel*;
- *remoteAccessNodes*: nodes involved in remote access tunnel. It contains all the nodes which have at least one security association in which the *source* network of the traffic is not defined `OR` the *destination* network is not defined;

**Algorithm 2** Translator algorithm

**Input:** $nVpnGw > 0$ ▷ Number of VPNGateway
1: *list siteToSiteNodes* ▷ Nodes involved in site to site tunnel
2: *list endToEndNodes* ▷ Nodes involved in end to end tunnel
3: *list remoteAccessNodes* ▷ Nodes involved in remote access tunnel
4: **for all** *node in nodes* **do**
5: **if** *source* $\neq$ *startChannel* AND *destination* $\neq$ *endChannel* **then**
6: *siteToSiteNodes* $\leftarrow$ *node*
7: **end if**
8: **if** *source* == *startChannel* AND *destination* == *endChannel* **then**
9: *endToEndNodes* $\leftarrow$ *node*
10: **end if**
11: **if** *source* == *any* OR *destination* == *any* **then**
12: *remoteAccessNodes* $\leftarrow$ *node*
13: **end if**
14: **end for**
15: **if** *siteToSiteNodes.size* $\geq 2$ **then**
16: SITETOSITE(*siteToSiteNodes*)
17: **end if**
18: **if** *endToEndNodes.size* $\geq 2$ **then**
19: ENDTOEND(*endToEndNodes*)
20: **end if**
21: **if** *remoteAccessNodes.size* $\geq 1$ **then**
22: REMOTEACCESS(*remoteAccessNodes*)
23: **end if**

The algorithm aims to analyze the operation of a VPN Gateway for different types of tunnels (as seen in 2). It takes various inputs, including the number of

VPN Gateways, and lists of nodes involved in site-to-site, end-to-end, and remote access tunnels.

- It initializes empty lists for site-to-site, end-to-end, and remote access nodes.
- For each node in the list of nodes:
  - If the node is not the start or end channel, it is considered as part of the site-to-site tunnel and added to the site-to-site nodes list.
  - If the node represents both the start and end channels, it is included in the end-to-end tunnel and added to the end-to-end nodes list.
  - If the node has "any" as the source or destination, it is identified as part of the remote access tunnel and added to the remote access nodes list.
- After processing all nodes, the algorithm evaluates the number of nodes in each category:
  - If there are at least two nodes in the site-to-site nodes list, the siteToSite function is executed. This function handles the configuration and management of the site-to-site tunnel.
  - Similarly, if there are at least two nodes in the end-to-end nodes list, the endToEnd function is executed. This function deals with the setup and maintenance of the end-to-end tunnel.
  - Lastly, if there is at least one node in the remote access nodes list, the remoteAccess function is executed. This function focuses on the configuration and handling of the remote access tunnel.

The procedures have the job to link the configuration of the security associations of the nodes to a *Strongswan* configuration.

The pseudo-code of the procedures is shown in Algorithms 3, 4 and 5.

These procedures are part of the implementation to handle the specific network configurations. The three algorithms act as follows:

- For first it is needed to sanitize all the inputs like source port, destination port, protocol and algorithms used for security enforcement;
- Then a file is created using the node's local name (i.e. its IP address);
- Now it is possible to write all the sanitized fields to the file;
- Iterate for each node in the list;

All the algorithms works in a similar way except for the data they refer to. In particular, algorithm 3 iterates over each node in the siteToSiteNodes list, algorithm 4 handles the nodes involved in EndToEndNodes list and algorithm 5 deals with the nodes in remoteAccessNodes list. This last one acts a bit differently, in fact it also creates two separate files, one for the local gateway (writeGW) and one for the remote host (writeH), because they contain different information.

**Algorithm 3** SiteToSite procedure

```
1: procedure SiteToSite(siteToSiteNodes)
2:     for all node ∈ siteToSiteNodes do
3:         srcPort ← sanitizePort(node.getSrcPort)
4:         dstPort ← sanitizePort(node.getDstPort)
5:         proto ← sanitizeProto(node.getProtocol)
6:         encAlgs ← sanitizeAlgs(node.getEncryptionAlgs)
7:         authAlgs ← sanitizeAlgs(node.getAuthenticationAlgs)
8:         file ← createFile(node.getLocalName)
9:         writeFile(localAddr, src, dst, srcPort, dstPort, authAlgs,
       encAlgs, remoteName)
10:    end for
11: end procedure
```

**Algorithm 4** EndToEnd procedure

```
1: procedure EndToEnd(endToEndNodes)
2:     for all node ∈ endToEndNodes do
3:         srcPort ← sanitizePort(node.getSrcPort)
4:         dstPort ← sanitizePort(node.getDstPort)
5:         proto ← sanitizeProto(node.getProtocol)
6:         encAlgs ← sanitizeAlgs(node.getEncryptionAlgs)
7:         authAlgs ← sanitizeAlgs(node.getAuthenticationAlgs)
8:         file ← createFile(node.getLocalName)
9:         writeFile(localAddr, src, dst, srcPort, dstPort, authAlgs, encAlgs,
       remoteName)
10:    end for
11: end procedure
```

**Algorithm 5** RemoteAccess procedure

```
1: procedure RemoteAccess(remoteAccessNodes)
2:     for all node ∈ remoteAccessNodes do
3:         srcPort ← sanitizePort(node.getSrcPort)
4:         dstPort ← sanitizePort(node.getDstPort)
5:         proto ← sanitizeProto(node.getProtocol)
6:         encAlgs ← sanitizeAlgs(node.getEncryptionAlgs)
7:         authAlgs ← sanitizeAlgs(node.getAuthenticationAlgs)
8:         file ← createFile(node.getLocalName)
9:         writeGW(localAddr, src, srcPort, protocol, authAlgs, encAlgs,
       localName)
10:        writeH(localAddr, dst, dstPort, protocol, authAlgs, encAlgs,
       remoteName)
11:    end for
12: end procedure
```

### 6.3.3 Naming the Nodes

Here is described the *Constructor* of the Java class `TranslatorStrongSwan`. To fulfill the interest of this work, each node configuration must contain a node name, which will be unique and representative.

Some "*name*" field already existed in the VEREFOO structure, in the node description and in the configuration description. Neither of them could be used due to some conflicts with past developments of the framework.

For this reason, a new field was introduced in the XML schema, called `vpnName`, which indicate the name of the node related to it. It is mandatory to fill this field, because the configuration file (see chapter 7.2) will contain values based on these names.

In listing 6.1 is showed an example of a host with "moon" as name, which is defined by the tag `<vpnName>`.

## 6.4 Input Sanitation and Translation

The authentication algorithm and the encryption algorithm must be *translated*, based on the corresponding wording for the algorithms declared in the Strongswan documentation.

In listing 6.2 is shown the translation of the Encryption Algorithms, while in listing 6.3 is shown the translation of the Authentication Algorithms.

Listing 6.2. Translation for Encryption algorithms

```
1  private String sanitizeEncryptionAlgorithm(String encAlg) {
2      switch(encAlg) {
3      case "TriploDES":
4          return "3des";
5      case "AES_128_CBC":
6          return "aes128";
7      case "AES_192_CBC":
8          return "aes192";
9      case "AES_256_CBC":
10         return "aes256";
11     case "AES_128_CTR":
12         return "aes128ctr";
13     case "AES_192_CTR":
14         return "aes192ctr";
15     case "AES_256_CTR":
16         return "aes256ctr";
17     case "CAMELIA_128_CBC":
18         return "camellia128";
19     case "CAMELIA_192_CBC":
20         return "camellia192";
21     case "CAMELIA_256_CBC":
```

```
22          return "camellia256";
23      case "CAMELIA_128_CTR":
24          return "camellia128ctr";
25      case "CAMELIA_192_CTR":
26          return "camellia192ctr";
27      case "CAMELIA_256_CTR":
28          return "camellia256ctr";
29      default:
30          throw new IllegalArgumentException("Invalid cipher suite
                value: " + encAlg);
31      }
32  }
```

Listing 6.3. Translation for Authentication algorithms

```
1  private String sanitizeAuthAlgorithm(String auth) {
2          switch(auth) {
3          case "MD_5":
4              return "md5";
5          case "MD_5_128":
6              return "md5_128";
7          case "SHA_1":
8              return "sha1";
9          case "SHA_1_160":
10             return "sha1_160";
11         case "SHA_2_256":
12             return "sha2_256";
13         case "SHA_2_384":
14             return "sha2_384";
15         case "SHA_2_512":
16             return "sha2_512";
17         default:
18             throw new IllegalArgumentException("Invalid cipher
                   suite value: " + auth);
19         }
20     }
```

# Chapter 7

# Validation and Testing

This chapter describes the software solution used to achieve the main goal of this thesis, to automatically configure VPN tunnels in Virtualized Network by use of *Strongswan*, earlier introduced in 6.

In the previous chapter, all the input were translated and made ready to be written in the Strongswan's configuration files, which is what this chapter is analyzing.

## 7.1 Introduction

This part of the thesis is focused on how has been possible to translate from the *Allocation Graph* VEREFOO gives in output to StrongSwan configuration files.

The allocation graph is written in medium-level language and describes network nodes and its configurations (as seen in 3) while the *swanctl.conf* file is written in low-level, proprietary language, which consist of hierarchical **sections** and a list of **key/value** pairs in each section. The structure of the file is better illustrated in 7.2.

As in the documentation of the tool [28], it is recommended to use the new style of configuration via the **vici control interface** and the **swanctl** command line tools. The **swanctl.conf** file, together with the *certificates* and *private keys* are stored in the *swanctl directory*.

The work of this thesis is to generate the configuration file, which is written in a proprietary low-level language, from the output of the VEREFOO framework. Therefore, the java class **TranslatorStrongSwan** has been completely rewritten, based on the new *vici plugin* and the *swanctl tool*.

This class examines the `XML` file given in input and extract the *Security Association* configurations, if any is present.

There are three main cases, already presented in chapter 2:

- **Site-to-Site**: this configuration enables the secure connection of two different networks. It makes possible to expands a network across geographically far

away offices and branches of a company or an institution connected using an insecure medium such as public internet.

- **End-to-End**: this configuration enables the secure connection among two host in a network.

- **Remote-Connection**: this configuration enables the secure connection from a host on the internet and a LAN, and vice versa. This kind of VPN provides access to the company network, which nowadays is very useful for all the remote workers.

## 7.2 The *swanctl.conf* file

The swanctl.conf file is a fundamental configuration file used by the Strongswan VPN solution. It serves as a **central configuration file** that governs the behavior and settings of the Strongswan's swanctl command-line tool. swanctl is responsible for managing the strongSwan IKE and IPsec configurations, allowing users to establish secure virtual private network (VPN) connections.

The swanctl.conf file plays a crucial role in defining the various aspects of the VPN configuration, including authentication methods, encryption algorithms, key exchange protocols, and network policies. It serves as a comprehensive and flexible **configuration framework** that enables users to tailor their VPN setup to meet their specific security and networking requirements. One of the notable advantages of using the swanctl.conf file is its ability to support both simple and advanced VPN configurations. It caters to a wide range of scenarios, from basic site-to-site or remote access VPNs to intricate setups involving multiple tunnels, diverse authentication methods, and fine-grained access control policies.

In this context, the swanctl.conf file acts as a **bridge** between the defined VPN configuration from VEREFOO and the Strongswan's underlying implementation. It provides a clear and structured format for specifying the necessary parameters and options, allowing users to define complex VPN setups with ease.

Understanding the structure and syntax of the swanctl.conf file is essential for effectively configuring and managing Strongswan VPN deployments. It requires careful consideration of the available options and parameters to ensure optimal security, performance and compatibility. In fact, this was one of the most important part of the study for this work.

The file is located in the `/etc/swanctl` folder of each machine where the software is loaded on. Examples of configurations of this file can be found in [29], while a more related one is listed in 7.1.

The parameters in the configuration file describe:

- *connections*: is the container holding the VPN connection;

- *local_addrs*: the address of the local gateway which is one end of the tunnel;

Listing 7.1. Moon swanctl.conf — Site-to-Site

```
1
2 connections{
3     site-site {
4       local_addrs = 93.10.168.1
5       remote_addrs = 93.10.168.2
6       local {
7          auth = pubkey
8          certs = moonCert.pem
9          id = moon.strongswan.org
10      }
11      remote {
12         auth = pubkey
13         id = sun.strongswan.org
14      }
15      children {
16         net-net {
17            local_ts = 10.0.0.0/24[tcp/22]
18            remote_ts = 10.0.1.0/24
19            start_action = trap|start
20            esp_proposals = aes128-sha2_256-modp2048
21         }
22      }
23   }
24 }
```

- *remote_addrs*: the address of the remote gateway which is the other end of the tunnel

- *local* section: information about the authentication method and certificates of the local gateway.

- *remote* section: information about the authentication method and certificates of the remote gateway.

- *children* section: where the child *Security Associations* parameter for the VPN connection are defined;

  - *local_ts*: list of local subnet to include in the SA. It must be a CIDR subnet definition. If none is defined, it takes the default value `dynamic` which match all the traffic. Moreover, it is possible to specify protocol and ports in the form `[protocol/port]`. They can be numeric or `getservent(3)` names;
  - *remote_ts*: list of subnet to include in the SA, as *local_ts*;
  - *start_action*: operation to carry out after the configuration loading, default=none; It is possible to trigger the tunnels as soon as traffic matching the rules is detected by the *trap* value, while *start* will initiate the connectivity as the service is started; They can be used simultaneously.
  - *esp_proposal*: algorithms proposals for the *Security Association*. It is a set of algorithms, they can be combined and presented in the form:

    `encriptionAlgs_integrityAlgs_DHGroups[optional]`

## 7.3 Validation

This section describes how the work was brought from theory to practice, showing how the algorithms defined above perform in some use cases.

### 7.3.1 Site-to-Site

In figure 7.1 is depicted the first case scenario which aims at securely interconnecting two VPN gateways. The authentication is based on `X.509` certificates.

In this case, a host in the subnet *MOON-NET* can ping a host in the subnet *SUN-NET*. The connection will be enciphered and authenticated from the VPN Gateway *Moon* to *Sun* and vice versa.

The `swanctl.conf` file of the *Moon* gateway can be found in listing 7.1, while the respective file for the *Sun* gateway is showed in listing 7.2.

Listing 7.2. Sun swanctl.conf — Site-to-Site

```
1
2  connections{
3      site-site {
4        local_addrs = 93.10.168.2
5        remote_addrs = 93.10.168.1
6        local {
7           auth = pubkey
8           certs = SunCert.pem
9           id = sun.strongswan.org
10       }
11       remote {
12          auth = pubkey
13          id = moon.strongswan.org
14       }
15       children {
16          net-net {
17             local_ts = 10.0.1.0/24
18             remote_ts = 10.0.0.0/24
19             start_action = trap|start
20             esp_proposals = aes128-sha2_256-modp2048
21          }
22       }
23    }
24 }
```

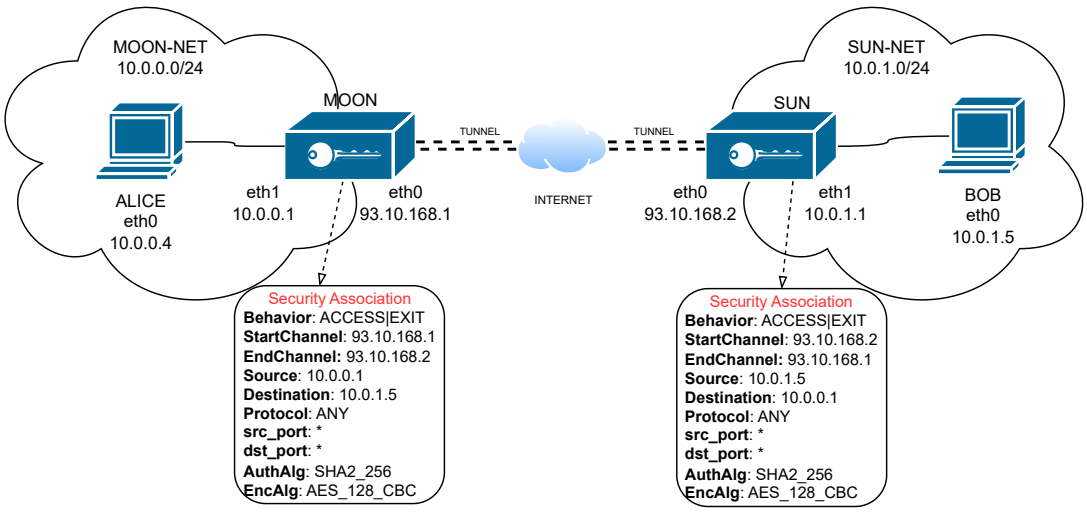


Figure 7.1. Site-to-Site scenario

### 7.3.2 End-to-End

In figure 7.2 is depicted the second case scenario which aims at describing the connection between two host with VPN capability. The authentication is based on `X.509` certificates.

As can be seen, the `StartChannel` and the `Source` addresses for the security association are the same. Simultaneously, the `EndChannel` is the same as the `Destination`. This happens because there is a `VPN Gateway` in each one of both nodes that handles the traffic sent and received by the node itself.

The `swanctl.conf` file of the *Alice* host can be found in listing 7.3, while the respective file for the *Bob* host is showed in listing 7.4.

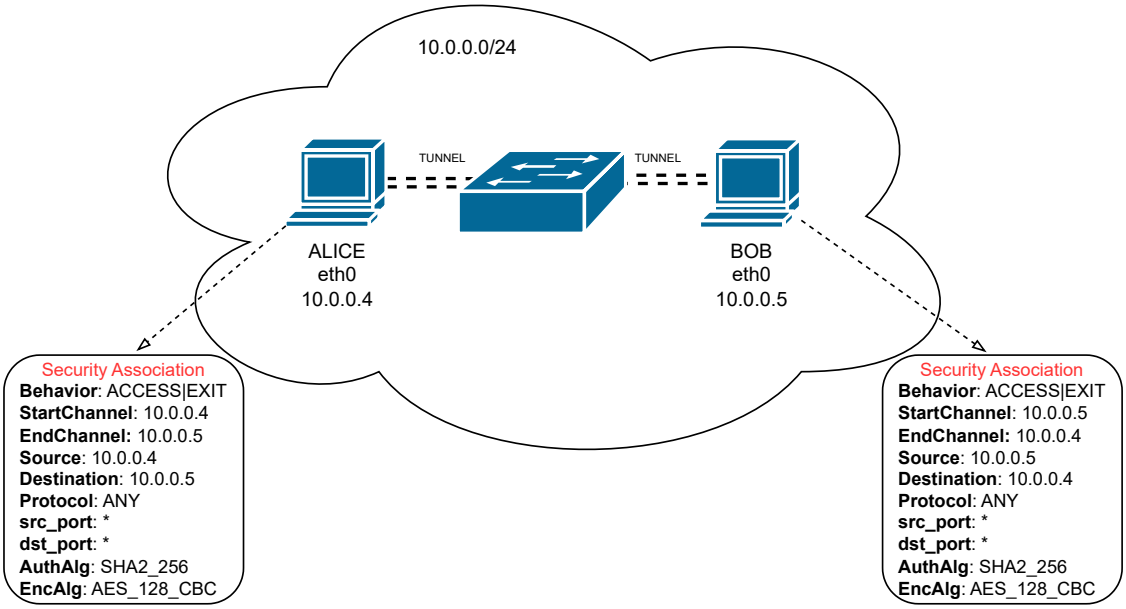


Figure 7.2. End-to-End scenario

Listing 7.3. Alice swanctl.conf — end-to-end

```
connections {
   end-end {
      remote_addrs = 10.0.0.5
      local {
         auth = pubkey
         certs = aliceCert.pem
      }
      remote {
         auth = pubkey
         id = C=CH, O=strongSwan, CN=bob.strongswan.org
      }
      children {
         end-end {
            start_action = trap|start
            esp_proposals = aes128-sha2_256-modp2048
         }
      }
   }
}
```

Listing 7.4. Bob swanctl.conf — end-to-end

```
connections {
   end-end {
      remote_addrs = 10.0.0.4
      local {
         auth = pubkey
         certs = bobCert.pem
      }
      remote {
         auth = pubkey
         id = C=CH, O=strongSwan, CN=alice.strongswan.org
      }
      children {
         end-end {
            start_action = trap|start
            esp_proposals = aes128-sha2_256-modp2048
         }
      }
   }
}
```

### 7.3.3 Remote-Connection

In figure 7.3 is depicted the third case scenario which aims at connecting a host with VPN capability to a remote VPN gateway. The authentication is based on `X.509` certificates.

In this case scenario, all the *Security Association* has been made explicit in order to fully show the configurations. As expected, for each node, the *Security Associations* are specular and in particular, it is always present the value `ANY` in at least one among `Source` and `Destination` fields.

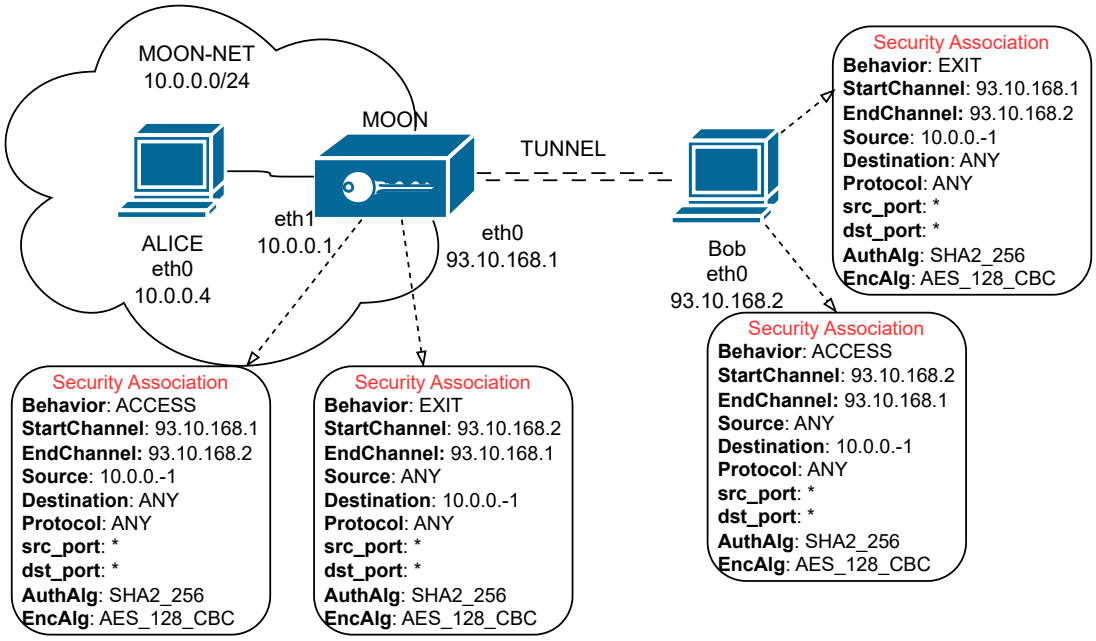


Figure 7.3. Remote-Connection scenario

The `swanctl.conf` file of the *Bob* host can be found in listing 7.6, while the respective file for the *Moon* gateway is showed in listing 7.5.

Listing 7.5. Moon swanctl.conf — Remote-Connection

```
connections {
   rc {
      local_addrs = 93.10.168.1
      local {
         auth = pubkey
         certs = moonCert.pem
         id = moon.strongswan.org
      }
      remote {
         auth = pubkey
      }
      children {
         net {
            local_ts = 10.0.0.0/24
            start_action = start|trap
            esp_proposals = aes128-sha2_256-modp2048
         }
      }
   }
}
```

Listing 7.6. Bob swanctl.conf — Remote-Connection

```
connections {
   home {
      local_addrs = 93.10.168.2
      remote_addrs = 93.10.168.1
      local {
         auth = pubkey
         certs = bobCert.pem
         id = bob.strongswan.org
      }
      remote {
         auth = pubkey
         id = moon.strongswan.org
      }
      children {
         home {
            remote_ts = 10.0.0.0/24
            start_action = start|trap
            esp_proposals = aes128-sha2_256-modp2048
         }
      }
   }
}
```

## 7.4 Testing

In this section is shown some test carried out to validate the algorithm of the translator.

For each host in the use case scenarios, a *Virtual Machine* has been created using `VirtualBox`. The creation and the configuration of all the machines is documented in Appendix A.

For the Translator to *push* the configuration files directly to the hosts, the Java Library `com.jcraft.jsch` has been used exploiting the `OpenSSH` project, the `SCP` protocol.

The code is listed in 7.7. For the purpose of the tests, `username` and `password` are simply deduced from the name of the hosts.

Others precautions taken, like write permissions for the remote `/etc/` folder, are specified in the Appendix A.

If the `Strongswan` service is running in the remote machine, with the execution of the function listed in 7.7, the swanctl.conf file is pushed to the right VPN Gateways and StronsgSwan is triggered to load the new configurations. Furthermore, thanks to the `start_action` policy specified in the swanctl.conf file (as seen in 7.2), if the tunnel failed to establish (but correctly configured), it will be started as soon as matching traffic will be detected.

Here follows a subsection illustrating the testing on some use case. Please, note that only the **ACCESS** type *Security Association* is listed. As in chapter 7.5, given the tunnels are always symmetrical, we do not need to consider the **EXIT** one. Despite that, the framework *needs* the definition of an EXIT SA, at this stage, for legacy compatibility. Here it has been omitted for the sake of simplicity and lengthiness.

Listing 7.7. Push configuration file to host Java code

```
1  private void fileUpload(String username, String remoteHost,
       String password) throws JSchException, SftpException,
       IOException {
2    String remoteHostName = username; ChannelExec channel = null;
3    try {
4      jschSession = jsch.getSession(username, remoteHost);
5      jschSession.setPassword(password);
6      jschSession.setConfig("StrictHostKeyChecking", "no");
7      jschSession.connect(1000);
8      channelSftp = (ChannelSftp) jschSession.openChannel("sftp");
9      if(channelSftp == null) {
10         System.out.println("Push configuration file aborted!");
11         return;
12     }
13     channelSftp.setOutputStream(System.err);
14     channelSftp.connect();
15     channelSftp.cd("/etc/swanctl/");
16     String localFile = "./StrongSwanConf/" + remoteHostName +
           "/swanctl.conf";
17     String remoteDir = "/etc/swanctl/";
18     channelSftp.put(localFile, remoteDir + "swanctl.conf",
           ChannelSftp.OVERWRITE);
19     channelSftp.exit();
20
21     channel = (ChannelExec) jschSession.openChannel("exec");
22     channel.setCommand("swanctl --load-all");
23     ByteArrayOutputStream responseStream = new
           ByteArrayOutputStream();
24     channel.setErrStream(System.err);
25     channel.setOutputStream(responseStream);
26     channel.connect(100);
27     while(channel.isConnected()) {
28       try {
29           Thread.sleep(1000);
30       } catch (InterruptedException e) {
31           e.printStackTrace();
32       } }
33     String responseString = new
           String(responseStream.toByteArray());
34   } finally {
35       if(jschSession != null)
36         jschSession.disconnect();
37       if(channel != null)
38         channel.disconnect();
39     }
40 }
```

### 7.4.1 Testing Site-to-Site

Referring to the configuration depicted in figure 7.4 and listing B.1 they describe the tunnel that will be established among the two VPN Gateway, using the interfaces with IPs `192.168.57.3` and `192.168.57.7` to connect the two private networks `10.0.0.0/24` and `10.0.1.0/24`. Strongswan will **encapsulate** the packets matching this parameter and use the `ICMP` protocol. Here, the port `22` is specified in MOON policies, while all the port range is specified in SUN. Anyway, since *ICMP* does not use ports, they will be **ignored** by the *Translator*.

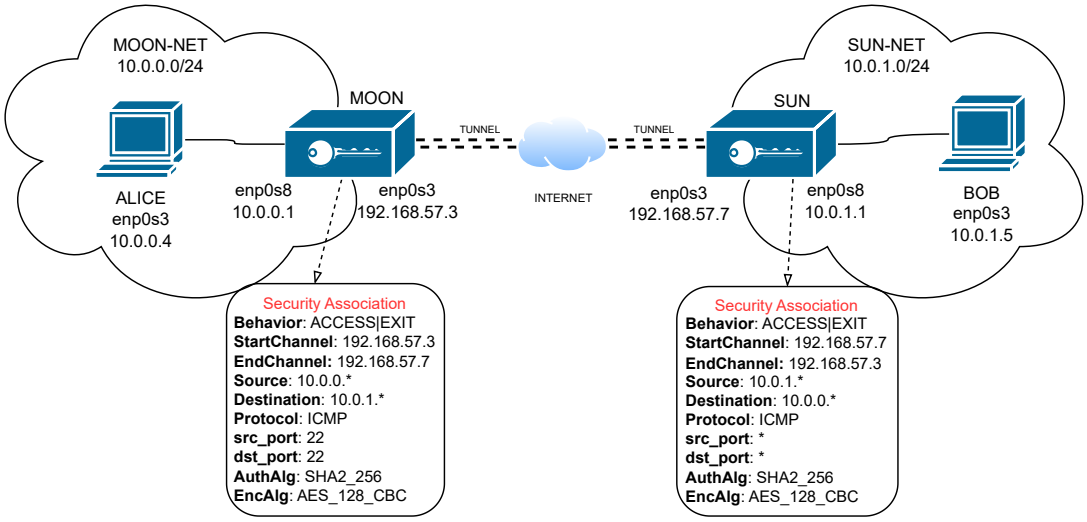


Figure 7.4. Testing Site-to-Site

Let us suppose the host Alice pings host Bob. They are in different subnets and are interconnected by a tunnel established between the VPN Gateway Moon and the VPN Gateway Sun. The *analyzer* is positioned in the interface `enp0s3` of Sun. The command executed is:

```
alice@Alice:# ping 10.0.1.5
```

The packet's traffic is shown in listing 7.8.

Listing 7.8. Packet capture ping command from Alice to Bob Site-to-Site

```
1  sun@sun:~$ sudo tcpdump -n -l --immediate-mode -i enp0s3 not
       port ssh and not port domain
2  tcpdump: verbose output suppressed, use -v[v]... for full
       protocol decode
3  listening on enp0s3, link-type EN10MB (Ethernet), snapshot
       length 262144 bytes
4  21:24:19.679565 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xca9d716b,seq=0xf), length 136
5  21:24:19.679565 IP 10.0.0.4 > 10.0.1.5: ICMP echo request, id
       16, seq 1, length 64
6  21:24:19.680249 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xc00e2448,seq=0xf), length 136
7  21:24:20.681706 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xca9d716b,seq=0x10), length 136
8  21:24:20.681706 IP 10.0.0.4 > 10.0.1.5: ICMP echo request, id
       16, seq 2, length 64
9  21:24:20.682450 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xc00e2448,seq=0x10), length 136
10 21:24:21.683109 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xca9d716b,seq=0x11), length 136
11 21:24:21.683109 IP 10.0.0.4 > 10.0.1.5: ICMP echo request, id
       16, seq 3, length 64
12 21:24:21.683796 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xc00e2448,seq=0x11), length 136
13 ^C
```

### 7.4.2 Testing End-to-End

Referring to the configuration depicted in figure 7.5 and listing B.2 they show that the tunnel will be established between the two end hosts `192.168.57.3` and `192.168.57.7`. Strongswan will be triggered when traffic matching the policies will be sent between them. Here all the port in range [`0-65535`] and all kind of protocols are specified.

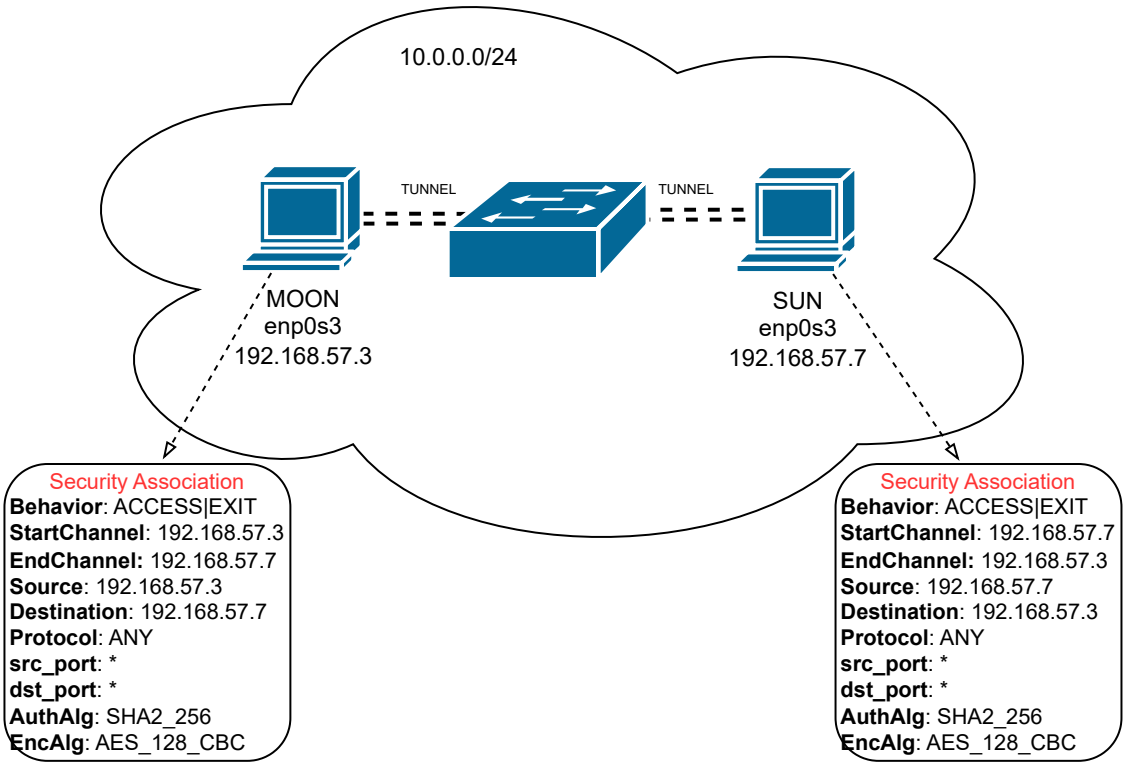


Figure 7.5. Testing end-to-end

Let us suppose the End host Moon pings the End host Sun. The *analyzer* is positioned in the interface `enp0s3` of Moon. The command executed is:

```
sun@Sun:# ping 192.168.57.3
```

The packet's traffic is shown in listing 7.9.

Listing 7.9. Packet capture ping command from Moon to Sun End-to-End

```
1  moon@moon:~$ sudo tcpdump -l --immediate-mode -i enp0s3 not port
       ssh and not port domain
2  tcpdump: verbose output suppressed, use -v[v]... for full
       protocol decode
3  listening on enp0s3, link-type EN10MB (Ethernet), snapshot
       length 262144 bytes
4  23:54:32.295799 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xcd5d5890,seq=0x5), length 136
5  23:54:32.295799 IP 192.168.57.7 > 192.168.57.3: ICMP echo
       request, id 13, seq 1, length 64
6  23:54:32.295919 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xc8da48fc,seq=0x5), length 136
7  23:54:33.297321 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xcd5d5890,seq=0x6), length 136
8  23:54:33.297321 IP 192.168.57.7 > 192.168.57.3: ICMP echo
       request, id 13, seq 2, length 64
9  23:54:33.297478 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xc8da48fc,seq=0x6), length 136
10 ^C
```

### 7.4.3 Testing Remote Connection

Referring to the configuration depicted in figure 7.6 and listing B.3 one can see the MOON-NET `10.0.0./24` can be accessed by the MOON VPN Gateway, which can serves numerous VPN clients. In this case the client BOB `192.168.57.7`, outside the MOON-NET wants to access it and establish a tunnel with the VPN Gateway.

Let us suppose the host Bob pings host Alice. They are in different networks and are interconnected by a tunnel established between the VPN Gateway Moon and the end node BOB. The *analyzer* is positioned in the interface `enp0s3` of the VPN Gateway MOON. The command executed is:

```
bob@Bob:# ping 10.0.0.4
```

The packet's traffic is shown in listing 7.10.

Listing 7.10. Packet capture of ping command from Bob to Alice Remote-Connection

```
1  moon@moon:~$ sudo tcpdump -n -l --immediate-mode -i enp0s3 not
       port ssh and not port domain
2  tcpdump: verbose output suppressed, use -v[v]... for full
       protocol decode
3  listening on enp0s3, link-type EN10MB (Ethernet), snapshot
       length 262144 bytes
4  02:42:28.530092 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xcefef0e3,seq=0x12), length 136
5  02:42:28.530092 IP 192.168.57.7 > 10.0.0.4: ICMP echo request,
       id 19, seq 1, length 64
6  02:42:28.530912 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xcc8512be,seq=0x12), length 136
7  02:42:29.531445 IP 192.168.57.7 > 192.168.57.3:
       ESP(spi=0xcefef0e3,seq=0x13), length 136
8  02:42:29.531445 IP 192.168.57.7 > 10.0.0.4: ICMP echo request,
       id 19, seq 2, length 64
9  02:42:29.532172 IP 192.168.57.3 > 192.168.57.7:
       ESP(spi=0xcc8512be,seq=0x13), length 136
10 ^C
```

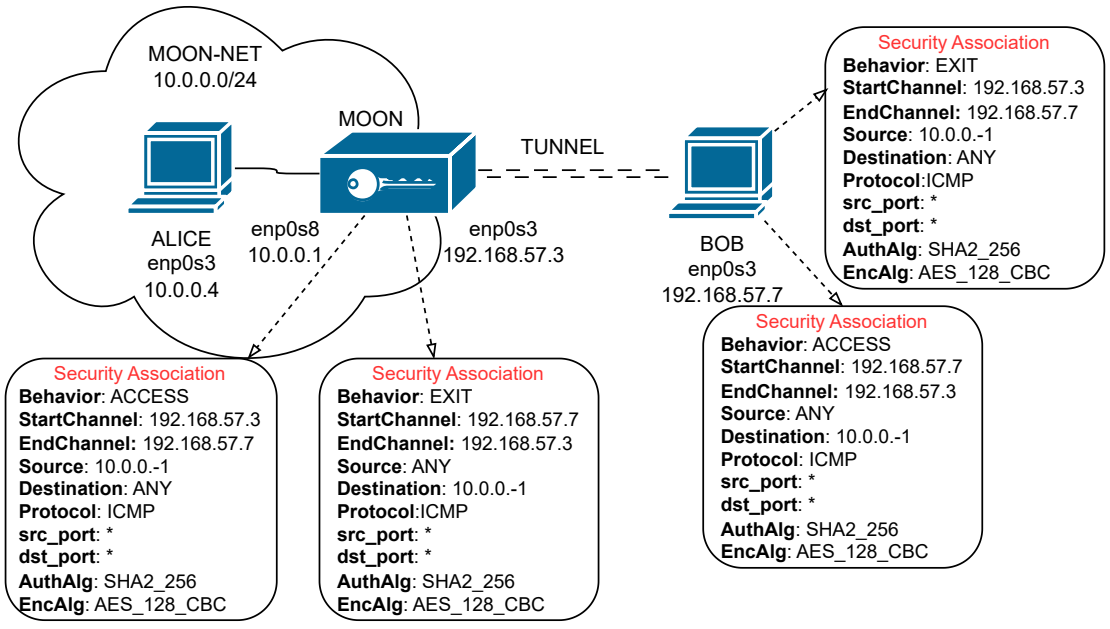


Figure 7.6. Testing Remote-Connection

From listing 7.8, 7.9 and 7.10, respectively for Site-to-Site, End-to-End and Remote-Connection, it is possible to see the traffic between the two VPN Gateways. Two kind of packets can be noticed:

- **ESP packets**: these are the packets exchanged through the tunnel. They are bigger than standard ICMP packets because of the extra ESP header.
- **ICMP packets**: these are the regular ICMP packets generated by the ping command; at first sight, one can think that the tunnel is not working as expected. In practice, it is working, but the packets are captured by the analyzer which is the node itself and, for these reasons, it will capture also the unencrypted packets.

## 7.5 Advantages and Limitations

In this section the analysis of pros and cons of this approach it is examined, in order to highlight all the useful feature added and focus on limitation that will inspire the future work.

It is straightforward to notice that with this approach to automatically configure the hosts in the network, there is the possibility to save time and resources with few computations, making it very useful. Moreover, it is *integrated* into the VEREFOO framework, and not shipped as a *plug-in*; this means all the configuration will be verified and correct by construction.

Unfortunately, this comes with some limitations, most of them coming from the *schema* of the network that *VEREFOO* uses as input (and gives in output).

It contains an high-level abstraction of the network and contains too few information about the real topology, like the one physically used for this work. For example, it was not possible to define subnet masks for the *IP Addresses*, For these reasons, all the examples and tests have been conducted with `/24` subnet mask as default.

Also, the double specification of Security Association in the XML schema of a VPN Gateway divided in ACCESS and EXIT is superfluous in this case, because as seen in 6.3.1 the tunnel will always be symmetrical and specular.

However, these limits are not unbridgeable, and a few fields of the schema can be easily added in the future.

# Chapter 8

# Conclusions and Future Work

During this thesis work, some additional feature of the VEREFOO framework has been developed and tested, with the purpose to extend the actual functionalities of the framework itself. This has been done with the purpose to exploit the already present modules to improve the effectiveness in a real NFV case scenario.

Firstly, a complete study of the framework and the field where it operates has been carried out, including the past work on how Virtual Private Network were introduced into the framework using the Atomic Flow and Maximal Flow approaches.

The focus of this work moved finally to the implementation of a Translator which is in charge of create configuration file for the VPN Gateways present in the network and push them on, using `scp` and `Strongswan`.

After the identification of the correct technologies to use, the Translator model was developed, together with some test to validate `Strongswan` new configuration files and automatic tunnel establishment from remote machines.

Finally, everything was put together and some test has been carried out, to validate the possibility to use the new tool extensively.

The improving of the representation of the nodes on the `XML` *schema* of VEREFOO, to meet the new requirements for this kind of side-developed tools is for sure one of the main future work that can be carried out.

In fact, it is needed a more in depth description of algorithm to use, IP addresses and netmasks to configure the nodes with.

Another big possible improvement is represented by the Strongswan API, that could not be used in this work based on Java. In fact, create a Python script to exploit the API could be very useful for the sake of security and simplicity to interact with the software.

The `IPSec` VPN software solution used in this work, `Strongswan 5.9.9`, from version `6.0.0` will use *OpenSSL* as new default Cryptographical Plugin making the *pki tool* used to create certificates and key pair (as described in appendix A) deprecated. One of the future work will be to use the recommended *OpenSSL* for the generation of the certificates.

# Bibliography


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

[2] D. Bringhenti, J. Yusupov, A. M. Zarca, F. Valenza, R. Sisto, J. B. Bernabé, and A. F. Skarmeta, "Automatic, verifiable and optimized policy-based security enforcement for sdn-aware iot networks," *Comput. Networks*, vol. 213, p. 109123, 2022. [Online]. Available: https://doi.org/10.1016/j.comnet.2022.109123

[3] D. Bringhenti, F. Valenza, and C. Basile, "Toward cybersecurity personalization in smart homes," *IEEE Secur. Priv.*, vol. 20, no. 1, pp. 45–53, 2022. [Online]. Available: https://doi.org/10.1109/MSEC.2021.3117471

[4] G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "A framework for verification-oriented user-friendly network function modeling," *IEEE Access*, vol. 7, pp. 99 349–99 359, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2929325

[5] 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

[6] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "A novel approach for security function graph configuration and deployment," in *7th IEEE International Conference on Network Softwarization, NetSoft 2021, Tokyo, Japan, June 28 - July 2, 2021*, K. Shiomoto, Y. Kim, C. E. Rothenberg, B. Martini, E. Oki, B. Choi, N. Kamiyama, and S. Secci, Eds. IEEE, 2021, pp. 457–463. [Online]. Available: https://doi.org/10.1109/NetSoft51509.2021.9492654

[7] D. Bringhenti, R. Sisto, and F. Valenza, "A novel abstraction for security configuration in virtual networks," *Computer Networks*, vol. 228, p. 109745, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128623001901

[8] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Towards a fully automated and optimized network security functions orchestration," in *2019 4th International Conference on Computing, Communications and Security (ICCCS), Rome, Italy, October 10-12, 2019*. IEEE, 2019, pp. 1–7. [Online]. Available: https://doi.org/10.1109/CCCS.2019.8888130

[9] S. Bussa, R. Sisto, and F. Valenza, "Security automation using traffic flow modeling," in *8th IEEE International Conference on Network Softwarization,*

*NetSoft 2022, Milan, Italy, June 27 - July 1, 2022*, A. Clemm, G. Maier, C. M. Machuca, K. K. Ramakrishnan, F. Risso, P. Chemouil, and N. Limam, Eds. IEEE, 2022, pp. 486–491. [Online]. Available: https://doi.org/10.1109/NetSoft54395.2022.9844025

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

[11] 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

[12] ——, "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

[13] D. Bringhenti and F. Valenza, "Optimizing distributed firewall reconfiguration transients," *Comput. Networks*, vol. 215, p. 109183, 2022. [Online]. Available: https://doi.org/10.1016/j.comnet.2022.109183

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

[15] F. Valenza and M. Cheminod, "An optimized firewall anomaly resolution," *J. Internet Serv. Inf. Secur.*, vol. 10, no. 1, pp. 22–37, 2020. [Online]. Available: https://doi.org/10.22667/JISIS.2020.02.29.022

[16] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, "Improving the formal verification of reachability policies in virtualized networks," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 1, pp. 713–728, 2021. [Online]. Available: https://doi.org/10.1109/TNSM.2020.3045781

[17] Z. Fu, S. F. Wu, H. Huang, K. Loh, F. Gong, I. Baldine, and C. Xu, "Ipsec/vpn security policy: Correctness, conflict detection, and resolution," in *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2001.

[18] 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

[19] C. Chang, Y. Chiu, and C. Lei, "Automatic generation of conflict-free ipsec policies," in *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, ser. Lecture Notes in Computer Science, F. Wang, Ed., vol. 3731. Springer, 2005, pp. 233–246. [Online]. Available: https://doi.org/10.1007/11562436_18

[20] M. M. G. Sadeghi, B. M. Ali, H. Pedram, M. Dehghan, and M. Sabaei, "A new method for creating efficient security policies in virtual private network," in *Collaborative Computing: Networking, Applications and Worksharing, 4th International Conference, CollaborateCom 2008, Orlando, FL, USA, November 13-16, 2008, Revised Selected Papers*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, E. Bertino and J. B. D. Joshi, Eds., vol. 10. Springer / ICST, 2008, pp. 663–678. [Online]. Available: https://doi.org/10.1007/978-3-642-03354-4_49

[21] M. Rossberg, G. Schaefer, and T. Strufe, "Distributed automatic configuration of complex ipsec-infrastructures," *J. Netw. Syst. Manag.*, vol. 18, no. 3, pp. 300–326, 2010. [Online]. Available: https://doi.org/10.1007/s10922-010-9168-7

[22] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Short paper: Automatic configuration for an optimal channel protection in virtualized networks," in *CYSARM@CCS '20: Proceedings of the 2nd Workshop on Cyber-Security Arms Race, Virtual Event, USA, November, 2020*, L. Chen, C. J. Mitchell, T. Giannetsos, and D. Sgandurra, Eds. ACM, 2020, pp. 25–30. [Online]. Available: https://doi.org/10.1145/3411505.3418439

[23] 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

[24] C. Basile, D. Canavese, A. Lioy, and F. Valenza, "Inter-technology conflict analysis for communication protection policies," in *Risks and Security of Internet and Systems - 9th International Conference, CRiSIS 2014, Trento, Italy, August 27-29, 2014, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. López, I. Ray, and B. Crispo, Eds., vol. 8924. Springer, 2014, pp. 148–163. [Online]. Available: https://doi.org/10.1007/978-3-319-17127-2_10

[25] S. Niksefat and M. Sabaei, "Efficient algorithms for dynamic detection and resolution of ipsec/vpn security policy conflicts," in *24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010*. IEEE Computer Society, 2010, pp. 737–744. [Online]. Available: https://doi.org/10.1109/AINA.2010.99

[26] "Strongswan software," https://www.strongswan.org.

[27] L. L. Mattina, "Optimized configuration of network security policies," 2021. [Online]. Available: https://webthesis.biblio.polito.it/20416/

[28] "Strongswan documentation," https://docs.strongswan.org/docs/5.9.

[29] "Strongswan's swanctl.conf examples," https://www.strongswan.org/testing/testresults/ikev2/.

[30] "Virtualbox manager," https://virtualbox.org.

[31] "Strongswan certificates quickstart," https://docs.strongswan.org/docs/5.9/pki/pkiQuickstart.html.

[32] "Strongswan's swanctl directory," https://docs.strongswan.org/docs/5.9/swanctl/swanctlDir.html.

# Appendices

# Appendix A

# AppendixA - Configuration

In the Appendix this work aims at giving the opportunity to replicate the configurations used and the tests performed during this work.

## A.1 Virtual Environment

The software used for virtualization is VirtualBox [30] (available for Linux, Windows and MacOs) while the software used for the establishment of *Security Channels* and *Certificates* is Strongswan [26] (available for Linux, Windows, MacOs and mobile platform like Android and iOS).

All the tests have been executed on a Linux machine with an `Ubuntu` based distribution. While it is true that Strongswan is cross-platform, it is strongly advised to use a Linux Machine to replicate the test due to its more fine granular possibility to set configurations and execute troubleshooting.

Two **groups** of *Virtual Machine* have been created:

1. The **Certificate Authority**: this VM has been created with the sole purpose of generating *self-signed* certificates and act as a Certificate Authority for all the machines used for the tests. The use of this VM is not mandatory, in fact you can create self-signed certificates for each of the VMs involved on the machine itself or have a third party, external, CA. In this case it was preferred to keep thing detached, The VM is based on `Ubuntu Server 22.04`;

2. The **Hosts/VPN Gateways**: the image used is `Ubuntu Server 22.04`, because it does not come with a GUI and, from this point of view, is lighter than an Ubuntu Client. Moreover, it comes with some tool already installed, which will be useful later on.

For the first group, one VM is enough, while it is possible to create as many VMs as needed of the second group. Screenshot A.1 shows the VMs used for this work.


Figure A.1. VirtualBox Groups Configuration

## A.2 Installation of Strongswan

At this stage, it is useful to have a `NAT network` configuration for the VM you are installing Strongswan on, in order for the VM to access the internet. Later on, in A.4 it is explained how the network will be configured in the final configuration.

After the VM creation, in order to install the Strongswan software, it is needed to install some *dependencies*, because it is going to be built from the source code. In the case of an Ubuntu based distribution, an example of command can be: `sudo apt-get install -y <dependency>`

Complete command:

```
sudo apt-get install -y build-essential net-tools libgmp-dev
  pkg-config libsystemd-dev libssl-dev automake
```

1. *Download* the Strongswan binaries, replacing `x.y.z` with the version you want to download. At the time of this thesis work, the last stable version is `5.9.9`:

   ```
   wget https://download.strongswan.org/strongswan-x.y.z.tar.bz2
   ```

2. *Extract* the archive and move into the new extracted folder;

   ```
   tar xjf strongswan-x.y.z.tar.bz2; cd strongswan-x.y.z
   ```

3. *Configure* the binaries with this specific options:

```
foo@bar:~$ ./configure --prefix=/usr --sysconfdir=/etc
    --enable-systemd --enable-swanctl --enable-x509
    --with-systemdsystemunitdir=/lib/systemd/system
    --enable-charon --disable-stroke --disable-scepclient
    --enable-openssl --enable-updown
    --with-capabilities=native
```

4. *Make*: actual compilation of the source code

```
foo@bar:~\$ make && sudo make install
```

5. *Permissions*: in order for the remote machine, where VEREFOO is executed, to push the configuration files, it is mandatory that the folder containing all the strongswan configuration files is accessible.

```
foo@bar:~$ sudo chmod 647 /etc/swanctl/
foo@bar:~$ cd /etc/swanctl
foo@bar:~$ sudo chmod 755 bliss ecdsa pkcs* private rsa
foo@bar:~$ sudo rm /etc/swanctl/swanctl.conf
```

6. *Reduced Privileges*: to remotely reload the StrongSwan settings and activate tunnels each time a new `swanctl.conf` file is pushed, it is needed to drop capabilities and run the daemon as *non-root* user. To do so, edit `/etc/strongswan.conf` file as:

```
1 charon {
2     load_modular = yes
3     user = $your_user_here
4     plugins {
5         include strongswan.d/charon/*.conf
6     }
7 }
8 include strongswan.d/*.conf
```

7. *Enable and start* the service:

```
foo@bar:~\$ sudo systemctl enable strongswan
foo@bar:~\$ sudo systemctl start strongswan
```

This must be replicated on all the VMs which needs Strongswan installed.

## A.3 Certification Authority

The Certification Authority is the VM in charge of issuing all the certificates for the end hosts and the VPN Gateway. The CA Virtual Machine needs Strongswan installed because its plugin `pki tool` is used to generate certificates. As in 8,

this plugin will be deprecated in the near future and `OpenSSL` will be replacing it instead. For Strongswan installation, see A.2.

Here follows a brief example of how to generate all the required *Certificates*, all the references can be found in the Strongswan documentation [31].

In order to follow the steps illustrated below, it is suggested to configure all the VMs network card in internal mode, using a single internal network. For more details, refer to A.4.

As per the following commands, each file is created in a specific directory of the root `/etc/swanctl`. In A.1 the directories used in this work are listed, while for the rest of them the reference is in [32].

### A.3.1 Generating CA Certificate

The first thing to do is to create a *Self-Signed CA Certificate* for the Certification Authority itself, in order to be able to create Certificates for all the entities involved in the tests.

1. The command

```
foo@bar:~$ pki --gen --type ed25519 --outform pem >
   /etc/swanctl/x509/private/strongswanKey.pem
```

   is used to generate an elliptic Edwards-Curve cryptographic *private* key.

2. The complementary public key is issued on the form of **self-signed** CA Certificate with 10 years lifetime:

```
foo@bar:~$ pki --self --ca --lifetime 3652 --in
   strongswanKey.pem --dn "C=CH, O=strongSwan,
   CN=strongSwan Root CA" --outform pem >
   /etc/swanctl/x509ca/strongswanCert.pem
```

### A.3.2 Generating End Entity Certificates

Now that the Certificate Authority is set, it can issue certificates for the hosts. Here are described the steps to follow, with particular focus on what command is executed in CA and in the End Host or VPN Gateway, which it is called host from now on in this subsection.

1. In the host VM (i.e. Moon), execute this command to generate a new Ed25519 private key:

```
moon@Moon:~$ pki --gen --type ed25519 --outform pem >
   /etc/swanctl/private/moonKey.pem
```

2. Now it is possible to use the private key to create a Certificate Request to the CA:

```
moon@Moon:~$ pki --req --type priv --in
   /etc/swanctl/private/moonKey.pem --dn "C=CH,
   O=strongswan, CN=moon.strongswan.org" --san
   moon.strongswan.org --outform pem >
   /etc/swanctl/moonReq.pem
```

   The above command allows creating a `PKCS#10` request to be sent and signed by the CA.

3. It is possible to send the request file through the use of the `SCP Protocol`:

```
moon@Moon:~$ scp /etc/swanctl/moonReq.pem ca@CAIp:/home/ca/
```

4. On the Certificate Authority VM, it is now possible to issue the certificate:

```
ca@CA:~$ pki --issue --cacert
   /etc/swanctl/x509ca/strongswanCert.pem --cakey
   /etc/swanctl/private/strongswanKey.pem --type pkcs10
   --in /home/ca/moonReq.pem --serial 01 --lifetime 1826
   --outform pem > /etc/swanctl/x509/moonCert.pem
```

5. The just created certificate in CA VM must be sent back to the end entity who made the request:

```
ca@CA:~$ scp /etc/swanctl/x509/moonCert.pem
   moon@moonIp:/home/moon/
```

6. The last thing to do is to put the certificate in the right folder, which is `/etc/swanctl/x509` of moon End Host.

Table A.1. Strongswan's swanctl Directory

| Directory | Contents |
|---|---|
| *x509* | x509 certificates |
| *x509ca* | CA trusted certificates |
| *private* | Private keys |
| *pubkey* | Raw public keys |

## A.4 Network Configuration

Now it is possible to finally edit the network configuration of the VMs, choosing among the ones Virtualbox offers, as in table A.2. The goal is to replicate the configurations seen in 7.4.

In each VM which acts as VPN Gateway, two network card must be activated:

Table A.2. Comparison between VirtualBox's networks

| Mode | VM → Host | VM ← Host | VM1 ↔ VM2 |
|---|---|---|---|
| Host-Only | + | + | + |
| Internal | − | − | + |

- **Internal** network: this creates an internal network where just the VMs can see each other. It is useful to define the subnet behind the VPN Gateways.
- **Host-Only** network: to put in communication the Host machine and the VPN Gateways. Will be used to copy the configuration file from the Host to the Strongswan folder on the VM itself with `ssh` and it is the network where the tunnel will be created.

To create an **Host-Only network**, the *Host Network Manager* of Virtualbox is needed and must be configured as in figure A.2

**Internal** network, instead, has no `DHCP`, so it is mandatory to assign to each network interface card the right IP configuration as in listing A.1.

Through the command `ip a` it is possible to see the network interface's name. The subnet used in the test is the `10.0.0.0/24`.

Listing A.1. Set IP address

```
vm@vm1:~$ sudo ip addr add x.x.x.x/y dev enp0xx
vm@vm1:~$ sudo ip link set enp0xx up
```

In each VM which acts as a EndHost, just one network card must be activated, with **internal** network capabilities.

### A.4.1 Forwarding

Each VPN Gateway must be able to forward packets between its network interfaces. On Linux gateways:

`sysctl net.ipv4.ip_forward=1`

`sysctl net.ipv6.conf.all.forwarding=1`

This can be made permanent by writing the two commands in the `/etc/sysctl.conf` file.

For example, in *Site-to-Site* case depicted in 7.4 it must be enabled a forwarding rule on the two VPN Gateways, otherwise they would not know how to route packets.

In that specific case, an `ip route` must be specified:

MOON: `sudo ip route add 10.0.1.0/24 via 192.168.57.4`

SUN: `sudo ip route add 10.0.0.0/24 via 192.168.57.3`

Forwarding rules must be enabled sometimes also on clients.

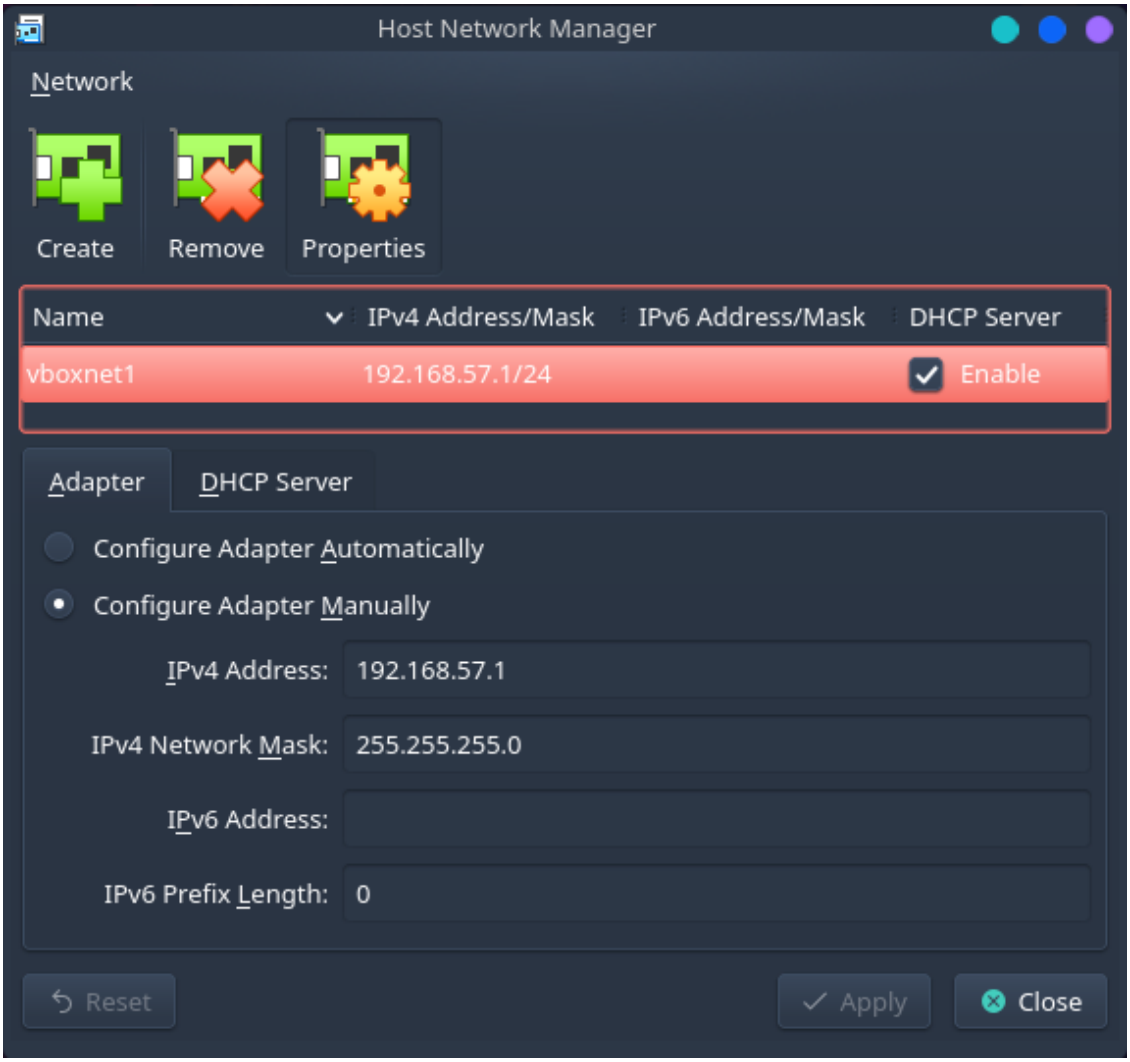


Figure A.2. Host Network Manager

Referring again to case 7.4, below the commands for Alice and Bob clients.

Alice: `sudo ip route add default via 10.0.0.1`

Bob: `sudo ip route add default via 10.0.1.1`

Now that network configuration and directory permissions are managed, the translator can be executed.

For an example of execution, refer to chapter 7.4.

# Appendix B

# AppendixB - XML schemas of test cases

## B.1 Site-to-Site

Listing B.1. XML Allocation Schema of a Site-to-Site case scenario

```
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
<graphs>
  <graph id="0">
   <node functional_type="WEBCLIENT" name="10.0.0.4">
     <neighbour name="192.168.57.3"/>
     <configuration description="A␣simple␣description" name="alice">
       <webclient nameWebServer="10.0.1.5"/>
     </configuration>
   </node>

   <node functional_type="WEBSERVER" name="10.0.1.5">
     <neighbour name="192.168.57.7"/>
     <configuration description="A␣simple␣description" name="bob">
       <webserver>
         <name>10.0.1.5</name>
       </webserver>
     </configuration>
   </node>

   <node functional_type="VPNGATEWAY" name="192.168.57.3">
     <neighbour name="10.0.0.4"/>
     <neighbour name="192.168.57.7"/>
     <configuration description="vpn␣gw␣access">
       <vpngateway>
         <vpnName>moon</vpnName>
         <securityAssociation>
             <behavior>ACCESS</behavior>
             <startChannel>192.168.57.3</startChannel>
             <endChannel>192.168.57.7</endChannel>
             <source>10.0.0.-1</source>
             <destination>10.0.1.-1</destination>
             <protocol>ICMP</protocol>
             <src_port>22</src_port>
```

```
            <dst_port>22</dst_port>
            <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
            <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm
          </securityAssociation>
        </vpngateway>
      </configuration>
    </node>

    <node functional_type="VPNGATEWAY" name="192.168.57.7">
      <neighbour name="10.0.1.5"/>
      <neighbour name="192.168.57.3"/>
      <configuration description="vpn␣gw␣exit">
        <vpngateway>
          <vpnName>sun</vpnName>
          <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel>192.168.57.7</startChannel>
            <endChannel>192.168.57.3</endChannel>
            <source>10.0.1.-1</source>
            <destination>10.0.0.-1</destination>
            <protocol>ICMP</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
            <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
            <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
          </securityAssociation>
        </vpngateway>
      </configuration>
    </node>
  </graph>
</graphs>
<Constraints>
  <NodeConstraints/>
  <LinkConstraints/>
</Constraints>
<PropertyDefinition>
  <Property graph="0" name="ProtectionProperty" src="10.0.0.4" dst="10.0.1.5"
    dst_port="80">
    <protectionInfo encryptionAlgorithm="AES_128_CBC"
        authenticationAlgorithm="SHA2_256">
      <untrustedNode node="20.0.0.3"/>
      <securityTechnology>TLS</securityTechnology>
      <securityTechnology>IPSEC</securityTechnology>
    </protectionInfo>
  </Property>
</PropertyDefinition>
</NFV>
```

## B.2 End-to-End

Listing B.2. XML Allocation Schema of a End-to-End case scenario

```
<NFV xsi:noNamespaceSchemaLocation="./xsd/nfvSchema.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<graphs>
    <graph id="0">
        <node name="192.168.57.3" functional_type="VPNGATEWAY"
            vpnCapabilities="true">
        <neighbour name="192.168.57.7" />
        <configuration name="AutoConf">
         <vpngateway>
            <vpnName>moon</vpnName>
            <securityAssociation>
                <behavior>ACCESS</behavior>
                <startChannel>192.168.57.3</startChannel>
                <endChannel>192.168.57.7</endChannel>
                <source>192.168.57.3</source>
                <destination>192.168.57.7</destination>
                <protocol>ANY</protocol>
                <src_port>0-65535</src_port>
                <dst_port>0-65535</dst_port>
                <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
                <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
            </securityAssociation>
            <securityAssociation>
                <behavior>EXIT</behavior>
                <startChannel>192.168.57.7</startChannel>
                <endChannel>192.168.57.3</endChannel>
                <source>192.168.57.7</source>
                <destination>192.168.57.3</destination>
                <protocol>ANY</protocol>
                <src_port>0-65535</src_port>
                <dst_port>0-65535</dst_port>
                <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
                <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
                </securityAssociation>
            </vpngateway>
        </configuration>
        </node>

        <node name="192.168.57.7" functional_type="VPNGATEWAY"
            vpnCapabilities="true">
        <neighbour name="192.168.57.3" />
        <configuration name="AutoConf">
        <vpngateway>
            <vpnName>sun</vpnName>
            <securityAssociation>
                <behavior>ACCESS</behavior>
                <startChannel>192.168.57.7</startChannel>
                <endChannel>192.168.57.3</endChannel>
                <source>192.168.57.7</source>
                <destination>192.168.57.3</destination>
                <protocol>ANY</protocol>
                <src_port>0-65535</src_port>
                <dst_port>0-65535</dst_port>
                <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
```

```
                <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
            </securityAssociation>
            <securityAssociation>
                <behavior>EXIT</behavior>
                <startChannel>192.168.57.3</startChannel>
                <endChannel>192.168.57.71</endChannel>
                <source>192.168.57.3</source>
                <destination>192.168.57.7</destination>
                <protocol>ANY</protocol>
                <src_port>0-65535</src_port>
                <dst_port>0-65535</dst_port>
                <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
                <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
            </securityAssociation>
        </vpngateway>
        </configuration>
        </node>

    </graph>
</graphs>
<Constraints>
    <NodeConstraints />
    <LinkConstraints />
</Constraints>
<PropertyDefinition>
    <Property name="ProtectionProperty" graph="0" src="192.168.57.3"
    dst="192.168.57.7" isSat="true">
    </Property>
</PropertyDefinition>
<ParsingString></ParsingString>
</NFV>
```

## B.3 Remote-Connection

Listing B.3. XML Allocation Schema of a Remote-Connection case scenario

```
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBCLIENT" name="10.0.0.4"
          vpnCapabilities="false">
        <neighbour name="192.168.57.3" />
        <configuration description="simple_description" name="alice">
          <webclient nameWebServer="10.0.0.1" />
        </configuration>
      </node>

      <node functional_type="VPNGATEWAY" name="192.168.57.3">
        <neighbour name="10.0.0.4" />
        <neighbour name="192.168.57.7" />
        <configuration description="vpn␣gw␣access" name="AutoConf">
          <vpngateway>
            <vpnName>moon</vpnName>
            <securityAssociation>
```

```
            <behavior>ACCESS</behavior>
            <startChannel>192.168.57.3</startChannel>
            <endChannel>192.168.57.7</endChannel>
            <source>10.0.0.-1</source>
            <destination>*</destination>
            <protocol>ICMP</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
            <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
            <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
          </securityAssociation>

          <securityAssociation>
            <behavior>EXIT</behavior>
            <startChannel>192.168.57.7</startChannel>
            <endChannel>192.168.57.3</endChannel>
            <source>*</source>
            <destination>10.0.0.-1</destination>
            <protocol>ICMP</protocol>
            <src_port>*</src_port>
            <dst_port>*</dst_port>
            <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
            <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
          </securityAssociation>

        </vpngateway>
      </configuration>
    </node>

    <node name="192.168.57.7" functional_type="VPNGATEWAY"
        vpnCapabilities="true">
      <neighbour name="192.168.57.3" />
      <configuration name="AutoConf">
        <vpngateway>
          <vpnName>bob</vpnName>
          <securityAssociation>
            <behavior>ACCESS</behavior>
            <startChannel>192.168.57.7</startChannel>
            <endChannel>192.168.57.3</endChannel>
            <source>*</source>
            <destination>10.0.0.-1</destination>
            <protocol>ICMP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>0-65535</dst_port>
            <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
            <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
          </securityAssociation>

          <securityAssociation>
            <behavior>EXIT</behavior>
            <startChannel>192.168.57.3</startChannel>
            <endChannel>192.168.57.7</endChannel>
            <source>10.0.0.-1</source>
            <destination>*</destination>
            <protocol>ICMP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>0-65535</dst_port>
```

```
            <authenticationAlgorithm>SHA2_256</authenticationAlgorithm>
            <encryptionAlgorithm>AES_128_CBC</encryptionAlgorithm>
          </securityAssociation>
        </vpngateway>
      </configuration>
    </node>

  </graph>
 </graphs>
 <Constraints>
  <NodeConstraints />
  <LinkConstraints />
 </Constraints>

 <PropertyDefinition>
  <Property graph="0" name="ProtectionProperty" src="192.168.57.7"
  dst="10.0.0.4" dst_port="80">
   <protectionInfo encryptionAlgorithm="AES_128_CBC"
      authenticationAlgorithm="SHA2_256">
    <securityTechnology>TLS</securityTechnology>
    <securityTechnology>IPSEC</securityTechnology>
   </protectionInfo>
  </Property>
 </PropertyDefinition>
</NFV>
```