# POLITECNICO DI TORINO

**Master's Degree in Computer Engineering**


**Master's Degree Thesis**

# A formal model of the capabilities for channel protection security controls

**Supervisor:**
**Prof. Cataldo Basile**

**Candidate:**
**Davide Colaiacomo**

**Academic Year 2023/2024**
**Torino**

# Abstract

In today's rapidly evolving cybersecurity landscape, professionals are tasked with managing numerous tools to safeguard systems against a growing array of threats. With diverse implementations of open-source and vendor-specific security controls, each utilizing its own configuration languages and ecosystems, selecting and disposing of the right solutions becomes a complex and time-consuming challenge for network administrators. This results, more often than desired, in human-based faults when security policies are enforced.

This thesis presents a formal model of security controls that abstracts security capabilities from a theoretical standpoint. By leveraging software engineering design patterns and best practices, this model enables the specification of security properties for information systems without requiring prior knowledge of the underlying enforcement technologies. A key feature of this approach has been the involvement of a software translator that seamlessly converts these high-level security requirements into low-level configurations specific to different technologies.

The research further explores the usage of this model to describe channel protection security policies from a high-level perspective, offering a new viewpoint on how abstract descriptions can drive practical security configurations. A significant part of this investigation is grounded in strongSwan, an IPsec solution widely used to secure remote network communication, which has served as a practical validation of the model.

This thesis demonstrates the model's capacity to produce reliable and robust security configurations by testing the framework in real-world cybersecurity scenarios. Moreover, its flexible architecture allows for future extensions, enabling support for a wide range of security software beyond the initial scope and expanding the possibilities for more adaptable cybersecurity solutions.

# Acknowledgements

Now that this experience at Politecnico di Torino is coming to an end, I would like to pause for a couple of minutes and reflect on the entire journey. I believe each goal in life is better when shared, so I want to express my gratitude to all the people who have supported and believed in me throughout the journey. First, I would like to dedicate a thought to my family, particularly my parents and brothers, who have always pushed me and encouraged me to follow my dreams and be ambitious no matter how hard it may seem. It is important to know that those who have seen you grow up (or have grown up with you) always have your back. Then, I want to thank all the friends that I have gotten to know during these years and before for making everything and every day of this long effort better. Without a doubt, this would not have been so pleasant and ended so positively if you had not stood by my side, as you all have done. Finally, I want to thank my academic supervisor, Prof. Cataldo Basile, for the opportunity to work with him, for his continuous feedback, and for his availability to help me overcome all my difficulties in the crowning work of this academic path. Thank you all for everything.

# Table of Contents

# List of Figures

# Acronyms

**AE** Authenticated Encryption

**AEAD** Authenticated Encryption with Associated Data

**AES** Advanced Encryption Standard

**AH** Authentication Header

**API** Application Programming Interface

**CA** Certificate Authority

**CBC** Cipher Block Chaining

**CFB** Cipher Feedback

**CRL** Certificate Revocation List

**CTR** Counter

**DM** Data Model

**DSA** Digital Signature Authentication

**DSCP** Differentiated Services Code Point

**IM** Information Model

**DES** Data Encryption Standard

**DH** Diffie-Hellman

**EAP** Extensible Authentication Protocol

**ECDH** Elliptic Curve Diffie-Hellman

**ECN** Explicit Congestion Notification

**ESP** Encapsulating Security Payload

**GCM** Galois Counter Mode

**GDPR** General Data Protection Regulation

**HMAC** Hash-based Message Authentication Code

**HSM** Hardware Security Module

**HTTP** HyperText Transfer Protocol

**ICV** Integrity Check Value

**IETF** Internet Engineering Task Force

**IKE** Internet Key Exchange

**IP** Internet Protocol

**IPC** Inter-Process Communication

**IPsec** Internet Protocol Security

**I2NSF** Interface to Network Security Functions

**LDAP** Lightweight Directory Access Protocol

**MAC** Message Authentication Code

**MTU** Maximum Transmission Unit

**NAT** Network Address Translation

**NFV** Network Function Virtualization

**NSF** Network Security Function

**OCSP** Online Certificate Status Protocol

**OOB** Out-Of-Band

**OOP** Object-Oriented Programming

**PFS** Perfect Forward Secrecy

**PKCS** Public-Key Cryptography Standards

**PKI** Public-Key Infrastructure

**POSET** Partially Ordered Set

**PPK** Post-Quantum Pre-Shared Key

**PSK** Pre-Shared Key

**QoS** Quality of Service

**RFC** Request For Comment

**RSA** Rivest-Shamir-Adleman

**SA** Security Association

**SDN** Software Defined Networking

**SHA** Secure Hash Algorithm

**SP** Security Policy

**SPD** Security Policy Database

**SPI** Security Parameter Index

**TCP** Transmission Control Protocol

**TPM** Trusted Platform Module

**TS** Traffic Selector

**TTL** Time-To-Live

**UML** Unified Modeling Language

**URI** Uniform Resource Identifier

**VICI** Versatile IKE Control Interface

**VPN** Virtual Private Network

**XMI** XML Metadata Interchange

**XML** Extensible Markup Language

**XSD** XML Schema Definition

# Chapter 1

# Introduction

In today's rapidly evolving technological landscape, organizations must frequently integrate and adapt to various solutions and platforms related to the cybersecurity domain. However, as new technologies are integrated into existing environments, managing the complexities of these systems becomes progressively more difficult; reliance on specific technologies has increased over the years, incrementing the risk of being locked to particular vendors and making it difficult for organizations to migrate to alternative solutions without significant costs or disruptions. Simultaneously, the growing sophistication of cyber threats places immense importance and pressure on security teams, enlarging the likelihood of human errors during the configuration, maintenance, and management of cybersecurity tools. These errors often result in severe security vulnerabilities, exposing systems to breaches, data theft, and even complete operational failures. Human intervention in many complex cybersecurity processes significantly amplifies the probability of mistakes.

The following sections will explore the consequences of *human errors* in cybersecurity and the phenomenon of *vendor lock-in*, highlighting the need to explore ways to mitigate these issues and improve overall security resilience.

## 1.1 Human errors in cybersecurity

*Human errors* in cybersecurity are significant for an organization's overall security. Despite advancements in technology and automation, human involvement remains critical in managing and securing information systems, as mistakes made by professionals and non-professionals can lead to severe vulnerabilities. Human errors often manifest in misconfigurations or oversights during the management of security measures, particularly for all those that require complex configuration steps. Moreover, employees might not fully understand and comply with security policies, leading to the adoption of risky practices.

Misconfiguring security tools or infrastructure usually means the person managing them makes wrong decisions and does not adopt adequate strategies and algorithms. This leads to systems inadvertently exposing data and leaving backdoors for attackers to use as they desire. Going more into the details of this scenarios, some consequences can be:

- *Financial loss*: breaches often lead to direct financial loss due to costs associated with remediation and user notification.
- *Reputational damage*: breaches can reduce customer and stakeholder trust, leading to lost business opportunities and a long-term decline in market reputation.
- *Regulatory consequences*: data protection laws like the *GDPR (General Data Protection Regulation)* impose fines for breaches that are proven to be consequences of negligence.
- *Operational inefficiency*: poor configuration of security tools might lead to false positives or excessive security alerts, overwhelming security teams and slowing response times to real threats.

History proves that *human errors* in cybersecurity are unavoidable and might always happen regardless of the professionals' expertise; still, solutions must be adopted to mitigate them as much as possible, for instance, through adherence to best practices and proper training, as the consequences can be enormous.

## 1.2 Vendor lock-in

In a technological environment, *vendor lock-in* refers to an organization becoming excessively dependent on a single vendor for products or services, making switching to an alternative provider difficult. This phenomenon can have significant implications for flexibility and innovation in practical scenarios, as transitioning from one vendor's products or services to another's incurs high costs and operational challenges. From a generic point of view, *vendor lock-in* can take several forms, such as:

- *Proprietary technologies*: many vendors use proprietary technologies that do not integrate easily with other systems; this can prevent organizations from moving data and processes to another platform.
- *Custom configurations*: some companies' tools may require extensive customization to align with the organization's needs; these customizations can be complex to replicate on new platforms, creating forms of dependency.
- *Data formats and storage*: many technological solutions often involve storing and processing large amounts of data; if this data is stored in proprietary formats, moving it to a new system without significant reformatting procedures can be challenging.

- *APIs and integration*: many platforms usually require integration with other parts of the organization's infrastructure; vendor-specific *API (Application Programming Interface)* can complicate migration to new platforms, as rebuilding an integration with a new vendor's *APIs* may be far from straightforward.

Diving deeper into the cybersecurity domain, *vendor lock-in* 's effects can be problematic due to the critical nature of security operations; the consequences can be multiple:

- *Inflexibility in Response to Threats*: if an organization is locked into a particular vendor, it may be unable to move quickly to new or improved security tools in response to emerging threats; for instance, if a new vulnerability affects a locked-in vendor's tool, the organization might fail to replace that tool with a better solution from another provider without incurring significant costs or downtime.

- *Reliance on a single vendor's security posture*: relying on a single vendor means that any vulnerability in that vendor's infrastructure could expose the organization; for instance, if a sophisticated attack targets a vendor, a lock-in situation implies the organization has fewer options to shift to another security provider to mitigate risk.

- *Lack of innovation*: vendors with locked-in customers might reduce their innovation incentives; over time, this can lead to stagnation in security practices and the use of outdated technologies, increasing an organization's risk of staying behind in the cybersecurity landscape.

- *Economic consequences*: moving away from a locked-in vendor typically involves time, financial investment, and resource allocation; for instance, this might manifest with new hardware and software purchases and staff retraining.

- *Reduced contractual power*: when organizations depend on a single vendor for critical security infrastructure, they may lose bargaining power in contract negotiations; as a consequence, the vendor may raise prices or reduce service quality, knowing the organization is unlikely to switch.

- *Lack of diversification*: in cybersecurity, diversification of tools and techniques can enhance defense mechanisms (this is known as *defense-in-depth*); relying on a single vendor can weaken this strategy, leading to a single point of failure.

Several reasons contribute to *vendor lock-in* in cybersecurity, mainly because many security tools are highly complex and specialized, leading to difficulties when organizations try to switch vendors once their systems are integrated. Moreover, the technical expertise required to operate specific security tools can be a barrier because, if the organization's cybersecurity team is accustomed to particular platforms, leading it on new tools can require significant time and effort.

## 1.3 Abstraction as a solution

From this general overview, it becomes clear that all these issues can be roughly linked by a common weakness, which is the excessive dependence on specific technological implementation; both the *vendor lock-in* and the *human errors* phenomena are caused by systems having to depend too heavily on the particular technology that is currently in use, causing difficulties in changing that same technology and to professionals becoming prone to errors in case they try to take this path.

*I2NSF (Interface to Network Security Functions)* working group proposed a solution to this problem by describing what different cybersecurity tools can perform regarding security functionalities; this has been achieved by realizing a model of these security functionalities following the principles of *abstraction* and *vendor-independence*. Previous thesis works have followed this approach; in particular, this thesis strongly references the work from *Cirella* 's thesis [1], with much of his results serving as a starting point for what has been achieved, and that will be thoroughly referenced in the following chapters.

In the context of this thesis, the principles mentioned above have been implemented focusing on the branch of **channel protection**, definable as the set of all the security elements and procedures that must be gathered and adequately utilized to guarantee secure communication between two or more parties when the data they exchange travels through possibly hostile mediums. The goal has been to leverage both modelling techniques already explored by previous works and new ones to explore the abstraction process of channel protection's implications; consequentially, the technology-independent aspects of this topic can be extracted, and professionals in real-world scenarios can be provided with a way to describe security configurations formally, but with the proper level of abstraction. The results consist of network administrators not being dependent on the actual channel protection implementation beneath; consequently, they can define security configurations without thinking about vendor's technologies and rely on a common abstract language to avoid human errors when switching from one low-level technology to another.

The structure of this thesis is defined as follows. `Chapter 2` will be dedicated to the theoretical background required to understand and carry out the work, focusing on the languages used during the design and implementation phases and the tools used for the validation phase. `Chapter 3` will focus on the related works that have been considered and studied for their approaches to the same topics. In `Chapter 4`, the modeling process for channel protection will be discussed, with great attention to the path that has led to the final and most suitable conception of a model usable in rigorous scenarios. Then, in `Chapter 5`, the actual implementation of this model will be presented, analyzing how it has been integrated into the already existing framework developed by *I2NSF* and previous thesis works. Finally, `Chapter 6` will describe the validation process for this implementation, underlying how the *strongSwan* tool, an *IPsec* famous implementation, has played a crucial role in this. `Chapter 7` will discuss eventual conclusions and possible future works that can start from these results.

# Chapter 2

# Background

This chapter will be dedicated to the theoretical background essential to understanding the issues this thesis addresses; all the topics introduced have been fundamental in the decision-making process for the design, implementation, and validation phases of the overall workflow, as will be underlined through the various chapters. In particular, having a generic but clear view of the *IPsec* protocol is helpful as an example of channel protection implementation and its strong relation with *strongSwan*. This validation tool will play a central role in later parts of this thesis, hence, it will be described thoroughly in this chapter. Moreover, the basics of the languages utilized for modeling and data representation will be recalled due to their extensive inclusion in all work steps.

## 2.1 IPsec

This section will present an overview of the *IPsec* standard to understand which problems the *strongSwan* project addresses. Note that this is not intended as a comprehensive analysis of *IPsec* but rather a summary of the most crucial aspects to grasp better the concepts presented in the thesis work. For more details about the single components of the standard, refer to the relative *RFC (Request For Comment)*, starting from the most recent about *IPsec* [2].

*IP (Internet Protocol)*, which is the most widely known protocol for network layer communication, does not inherently have any feature to guarantee the security of the data involved; as a consequence, **IPsec (Internet Protocol Security)** was defined to face this issue, and it is a framework used to protect *IP* traffic on the network layer by applying the security properties required for a secure exchange of data, namely *confidentiality*, *integrity*, *authentication* and *anti-replay*.

Consider that, in many real cases, the integrity property is automatically provided by performing the authentication of the data, so each time it is stated that authentication is provided, it can be assumed that integrity is provided as well, even if not explicitly stated. From the most general point of view, the idea is that two peers desire to communicate by exchanging *IP* packets; to protect these packets, the two peers need first to build an *IPsec* tunnel through which these packets can be securely exchanged, and this is obtained using the *IKE* protocol.

### 2.1.1 Internet Key Exchange

The **IKE (Internet Key Exchange)** protocol establishes a secure channel between two devices to transmit user data safely. Historically, there are two versions of *IKE*, which are *IKEv1* and *IKEv2*; the details about implementations for *IKEv1* will be presented just for completion purposes, as it has been deprecated for a few years now and is still supported only for legacy applications.

Regardless of the version, from a high-level perspective, it consists of a sequence of actions that aim at defining a set of security algorithms and parameters, known as **SA (Security Association)**, to protect the user *IP* data traveling from one device to another. Substantially, a *SA* is a way to enforce some guidelines programmed into the specific *IPsec* implementation, which go by the name of **SP (Security Policy)** and specify the possible options to process *IP* traffic from a security point of view. The *IKE* process can be split logically into two phases.

#### First phase

In the first phase of *IKE*, the peer who wishes to protect its traffic will initiate the protocol and will play the role of the *Initiator*, while the other peer is the *Responder*; the goal is to establish a secure channel through which the details about how to protect the *IP* traffic can be negotiated. Specifically, the items that get exchanged are a *hash algorithm* to provide integrity, an *authentication method* to prove one's identity, an *encryption algorithm* to provide confidentiality, a *DH (Diffie-Hellman) group* to determine the strength of the key used in the exchange process, and a *lifetime* to specify how long the tunnel will stand up. Once this negotiation is achieved, the two peers have negotiated a specific *SA* for future steps, known as **IKE_SA**, and the first tunnel is instantiated.

In *IKEv1*, the first phase is known as *IKE Phase 1* and all these steps can be performed through two different modes, which are *Main mode* and *Aggressive mode*; the former has a total of six messages exchanged between the peers, so the process is slower but more secure thanks to the identification data getting encrypted, while the latter has a total of only three messages exchanged between the peers, leading to a faster but less secure process as identification data are left in clear-text.

In *IKEv2*, the operative modes have been deprecated. The first phase, which is known as **IKE_SA_INIT**, requires only two messages to be completed (unless the *Respondent* desires to verify that it is not under a *Denial of Service* attack, in which case it will ask for an additional message with a *cookie* from the *Initiator*). Using the *Key Exchange (KE)* and the *Nonces (N)*, the peers can derive a *Shared Secret* to encrypt all the following *IKE* messages based on the *IKE_SA* negotiated via the *SA1* payloads.

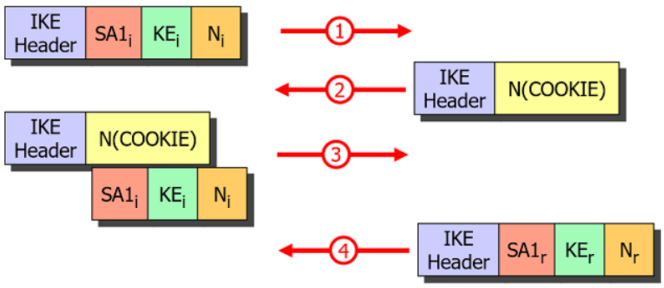


**Figure 2.1:** IKE_SA_INIT

### Second phase

In the second phase of the *IKE* protocol, the peers rely on the tunnel derived from the first phase to instantiate another one, commonly referred to as *IPsec tunnel*, that will be used to protect the actual user data.

To achieve so, they need to negotiate an *encryption algorithm* for confidentiality, an *authentication algorithm* for peer authentication, a *lifetime* to state how long the *IPsec* tunnel will be valid, an optional *DH exchange* to guarantee a property known as *PFS (Perfect Forward Secrecy)* (in brief, the communication keys are generated for each session so that, if the current key gets compromised, the past exchanges remain secure), and finally the *IPsec protocol* (*AH* or *ESP*) and *encapsulation mode* (*transport* or *tunnel*) that can be used and which will be discussed later.

In *IKEv1*, these steps are performed through a single mode known as *Quick mode*, which consists of a total of 3 messages; on the other hand, in *IKEv2*, the second phase is explicitly known as **CHILD_SA**, there is not a specific operative mode, and all are carried out in exchanges of two messages for each *CHILD_SA* that is defined or for rekey and informational purposes. The first pair exchange is also known as **IKE_AUTH** because its purpose is for the peers to prove their own identity to each other; the format of the two messages for the *IKE_AUTH* exchange varies depending on the adopted authentication method (some frequently used methods are *certification-based* authentication, as shown in the image below, *PSK-based* authentication and *EAP-based* authentication).

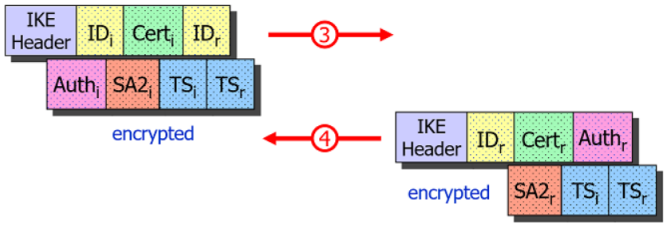


**Figure 2.2:** Certification-based Authentication

What is consistent among all the approaches is the fact that the *Initiator* proposes a *SA* and a set of *TS* to be used for the first actual *CHILD_SA*, and the *Responder* provides back a selected *SA* and a possibly narrowed set of *TS* so that all the necessary information to protect the *IP* packets is decided and the *IPsec* tunnel is instantiated. Subsequent pair exchanges, known as **CREATE_CHILD_SA**, can be executed to negotiate additional *CHILD_SA*s or to perform the periodic rekeying of either the *IKE_SA* or a *CHILD_SA*. In particular, the presence of the *N* notification leads to a *CHILD_SA* rekeying, while its absence is to perform the rekeying of the *IKE_SA*.

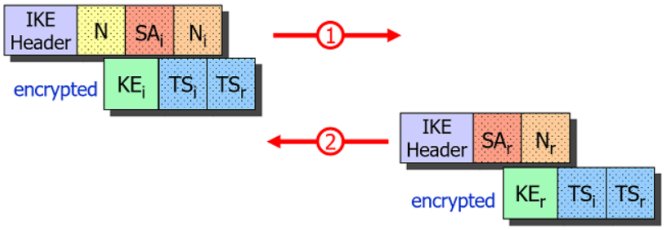


**Figure 2.3:** CREATE_CHILD_SA

### 2.1.2 IPsec protocols

Once the *IPsec* tunnel is instantiated, the data traveling through it will be protected with the algorithms negotiated during the *IKE* protocol; how this protection is concretely applied still has to be analyzed. As anticipated in the previous paragraph, the communicating peers, among other elements, negotiate the protocol through which the protection is applied, which can be *AH* or *ESP*, and the mode with which it is applied, which can be *transport* or *tunnel*.

Regarding the *IPsec* protocols, the **AH (Authentication Header)** protocol offers authentication of the *IP* packet but not encryption. Generally, *AH* protects the *IP* packet by calculating a hash value over the fields in the *IP* header, excluding those that can change in transit (namely *TTL (Time-To-Live)* and *header checksum*), which would otherwise fault the authenticity check at the destination. The **ESP (Encapsulating Security Payload)**, on the other hand, offers authentication and encryption of the *IP* payload but not of the *IP* header; despite so, it is more widely adopted than *AH*.

It should be considered that *ESP* was initially defined to provide only confidentiality of the payload, while the possibility to provide authentication has been added later; still, the algorithms used for authentication can be the same regardless of the protocol.

About the application modes for these protocols, adopting *IPsec* in **transport** mode requires that, starting from the original *IP* packet, an additional *IPsec* header (which might be *AH* or *ESP*) is inserted between the *IP* header and the payload, applying the requested security properties to the packet. Conversely, adopting *IPsec* in **tunnel** mode requires that, before using the additional header for security purposes, a tunnel is instantiated by encapsulating the original *IP* packet in a new *IP* header. Then, the *IPsec* header is inserted between the new *IP* header and the updated payload, which now also contains the original *IP* header.

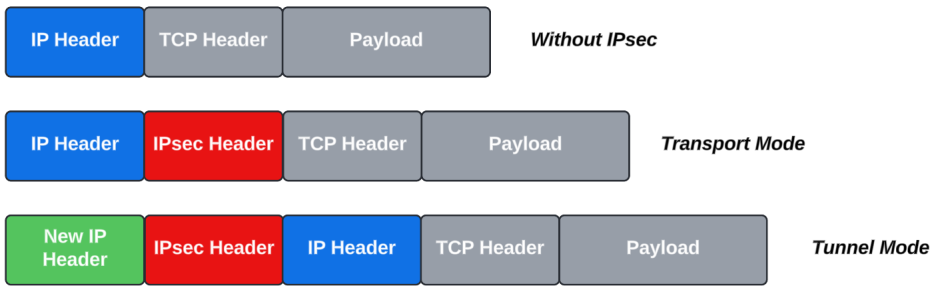


**Figure 2.4:** IPsec modes

It is to be noted that *transport* mode provides encryption (if supported by the protocol) only to the payload of the packet and not to the header, which might be acceptable only depending on the actual usage scenario; on the other hand, *tunnel* mode also secures the original header of the packet. Moreover, given the nature of the packet transformation, *transport* mode is mainly adopted to perform host-to-host secure communication. In contrast, *tunnel* mode is suited to connect networks securely.

### Authentication Header in transport mode

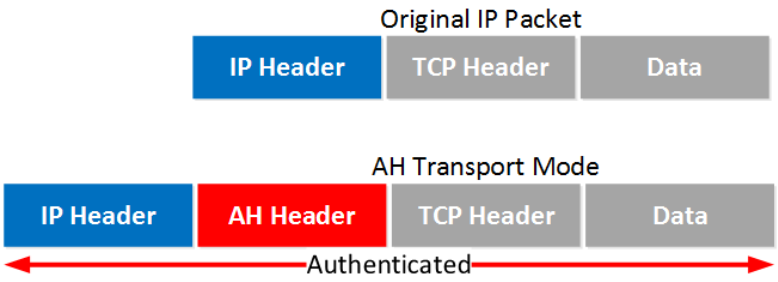


**Figure 2.5:** AH in transport mode

*AH*, used in *transport* mode, requires an *AH* header to be inserted into the original *IP* packet between the *IP* header and the payload. In this situation, the authentication property is provided to the whole packet. It can be verified at the *Receiver* side employing a procedure that requires a special value inserted in the *AH* header, known as *ICV (Integrity Check Value)*; if this procedure fails, the packet is usually discarded without further analysis.

### Authentication Header in tunnel mode

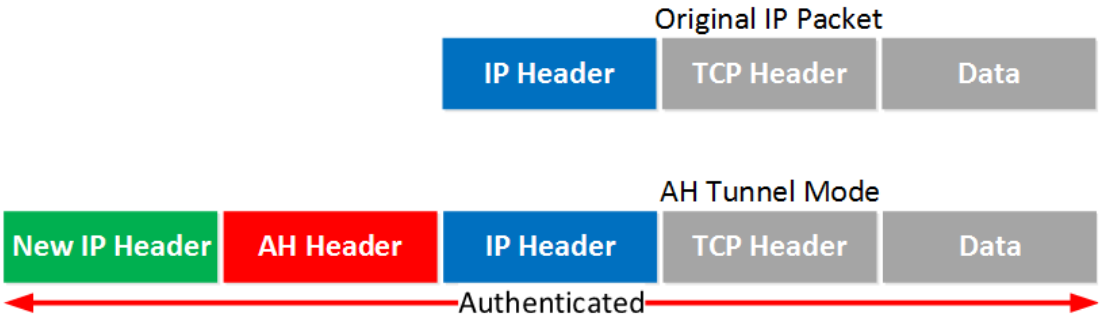


**Figure 2.6:** AH in tunnel mode

Adopting *AH* with *tunnel* mode, a new *IP* header is added to the original *IP* packet before applying the protocol; the *AH* header is then inserted between the new *IP* header and the new payload, which is the original *IP* packet. The authentication property is guaranteed to the whole packet, including the new *IP* header, while the validation details are the same of *transport* mode.

### Encapsulating Security Payload in transport mode

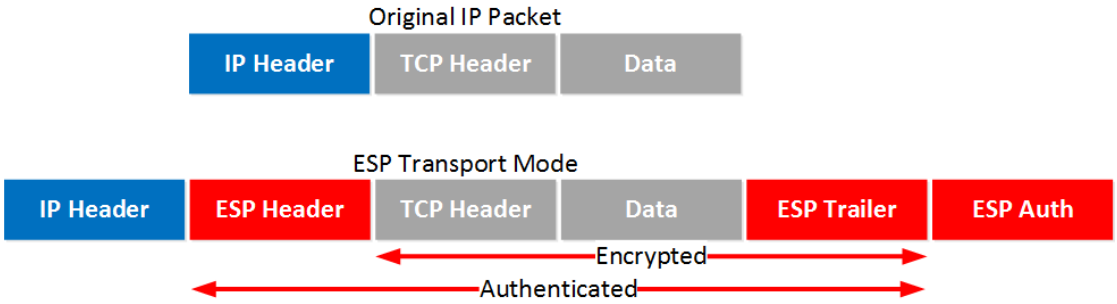


**Figure 2.7:** ESP in transport mode

The *ESP* protocol in *transport* mode requires that the original payload of the *IP* packet gets delimited by an *ESP* header, inserted between the payload and the original *IP* header, and an *ESP* trailer, appended after the payload, to guarantee the encryption of the original payload; moreover an *ESP* auth component is optionally added to provide authentication for the payload of the final packet.

### Encapsulating Security Payload in tunnel mode

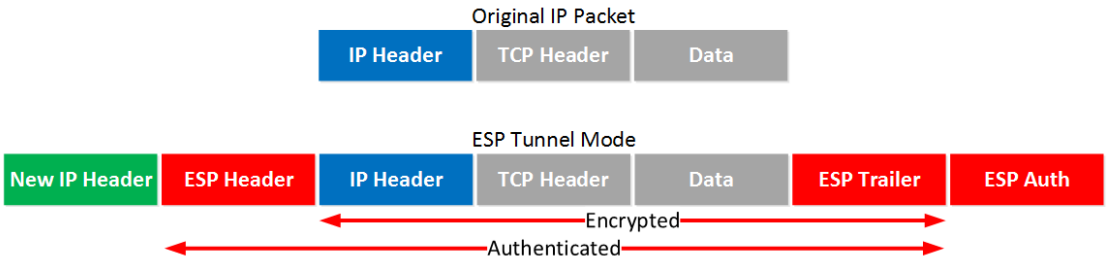


**Figure 2.8:** ESP in tunnel mode

Regarding *ESP* in *tunnel* mode, the original *IP* packet receives a new *IP* header and becomes the payload of a new *IP* packet and is then delimited by the *ESP* header and the *ESP* trailer and *ESP* auth components. The original *IP* header is also encrypted and authenticated.

### AH + ESP in transport mode

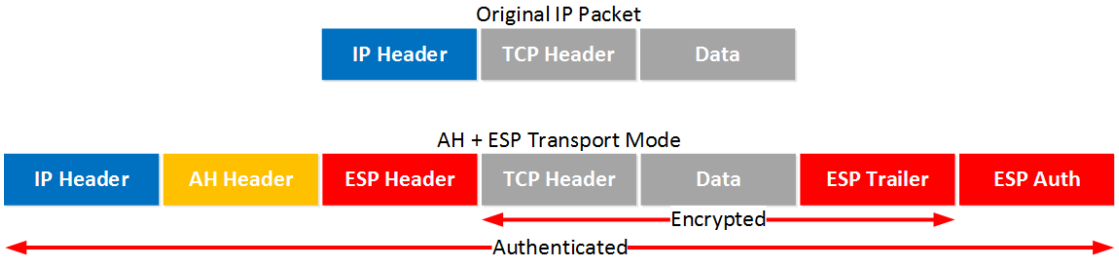


**Figure 2.9:** AH + ESP in transport mode

The two protocols can be combined in *transport* mode by taking the original *IP* packet and inserting both an *AH* header and an *ESP* header between the *IP* header and the payload while also adding the *ESP* trailer and *ESP* auth as required by *ESP*. Regarding the security properties applied to the packet, the result is similar to *ESP*'s. However, thanks to the *AH* header, authentication is provided to the entire packet.

### AH + ESP in tunnel mode

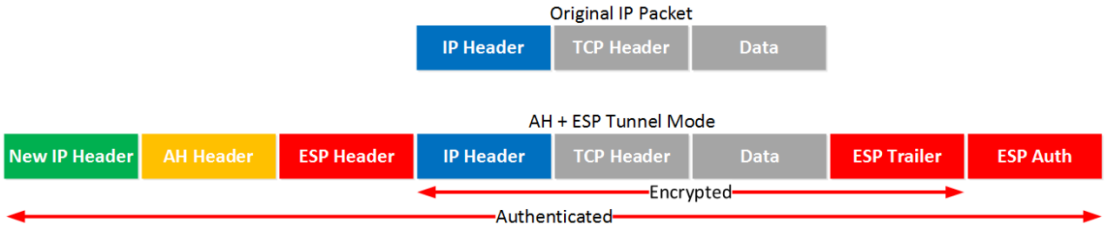


**Figure 2.10:** AH + ESP in tunnel mode

Combining the *AH* and *ESP* protocols in *tunnel* mode requires adding a new *IP* header to the original *IP* packet. An *AH* header and an *ESP* header are inserted between the new *IP* header and the original one, and the *ESP* trailer and *ESP* auth are appended to the whole packet. The original *IP* header and payload are encrypted, and authentication is provided to the final packet, including the new *IP* header.

## 2.2 System modeling and data representation

As already anticipated, extensive modeling had to be performed to implement abstract descriptions of channel protection. This step is crucial for understanding how this cybersecurity branch maintains a consistent and structured approach across different low-level tools and protocols. Once the modeling operations are concluded, the final model must be represented in a processable way by software applications relying on standard representation formats. This approach ensures that the abstract concepts are understandable to human stakeholders and seamlessly integrated and utilized by automated systems. This section will present an overview of the languages and formats used to model and represent the final system in machine-readable form.

### 2.2.1 UML

The **UML (Unified Modeling Language)** is a standardized visual language used to model and document software systems, architectures, and processes. It provides tools that describe the structure and behavior of software systems, making it easier to visualize and design complex applications. *UML* aims to provide a common language that can be understood across different domains, and it is widely used during various stages of software development, from requirements gathering and analysis to design and implementation; moreover, since it is not a programming language, it has garnered popularity as a way to convey descriptions of systems, which should eventually be implemented by programmers, to non-computer experts.

*UML* comprises various diagram types broadly categorized into two main groups, namely *structural* diagrams and *behavioral* diagrams. Structural diagrams represent the static aspects of a system and describe the elements that make it up, such as classes, objects, and packages, and the relationships between them. On the other hand, behavioral diagrams represent a system's dynamic aspects, which are how the system behaves during execution and how it reacts in response to specific stimuli. For this thesis work and the related ones that were previously developed, the most used type is the *class* diagram, which is a specialization of structural diagrams [3].

#### Class diagram

**Class** diagrams are a type of *UML* that is possibly the most widely employed in software development processes; it is used to document and perform design operations for object-oriented systems and aims at representing their structures through *classes* and *relationships*. These entities embody a visual approach suited for these systems; an overview of their most important aspects follows.

A **class** is a concept of *OOP (Object-Oriented Programming)* that represents the template for creating an *object*. An *object* is an instance of a *class*, and each class defines a group of attributes and operations that the object instantiated will possess; more precisely, attributes represent the properties of the *object*, while operations specify which actions the *object* can perform.

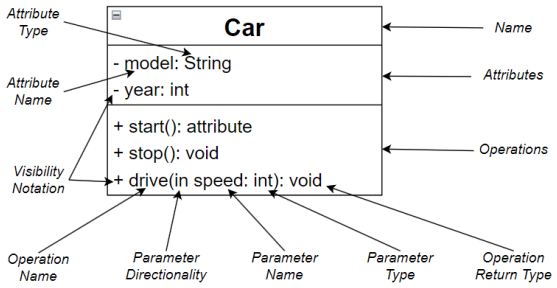


**Figure 2.11:** UML Class example

Going into the most ordinary details of the notation, each *class* has a *name* and, optionally, a set of *attributes* and a set of *operations*, as stated above. Each attribute has an *attribute name* and an *attribute type*, while each operation has an *operation name* and a *return type*. Moreover, each operation can be provided with one or more *parameters*; each *parameter* has a *parameter name* and a *parameter type* and is also provided with a *parameter directionality*. This last characteristic clarifies the flow of information between classes so that when an operation is invoked, it is understandable how data is passed; this directionality can be specified as *in* if the parameter carries input data from the caller to the called *object*, *out* if the parameter will eventually carry output data for the caller provided by the called *object*, and *inOut* if the parameter can serve both these purposes. Finally, each *attribute* and operation has a *visibility notation*, which indicates its access level. In particular, if it is visible to all *classes*, it is defined as *public* through the `+` marker; if it is visible only within the *class*, it is defined as *private* through the `-` marker; if it is visible to *sub-classes*, it is defined as *protected* through the `#` marker; if it is visible to all the *classes*, but in the same *package*, it is defined as *default* through the $\sim$ marker.

In *OOP*, traditional classes are also often related to *abstract* classes and *interfaces*. These concepts help enforce design principles and organize code by defining common behaviors across different classes while still allowing flexibility in implementation.

*Abstract* classes can combine both methods with concrete implementations and methods without implementation, known as *abstract methods*. Classes that extend an *abstract* class must provide concrete implementations for all of its *abstract methods* unless the extending class is also *abstract*. It is important to note that *abstract* classes cannot be instantiated directly; they require a subclass that provides implementations for the *abstract methods*. One fundamental limitation is that a traditional class can extend only one *abstract* class, adhering to the principle of single inheritance in many *OOP* languages like *Java* and *C++*.

On the other hand, *interfaces* are purely a collection of method declarations. They specify *abstract methods* but do not provide any concrete implementations for these methods. Unlike *abstract* classes, *interfaces* cannot have any method implementations or non-static fields. All fields defined in an *interface* are implicitly *public static final*, making them constants. A traditional class that implements an *interface* must provide implementations for all declared methods. However, an important distinction is that a class can implement multiple *interfaces*, which allows for a more flexible and modular design, enabling a form of multiple inheritance by combining behaviours from various interfaces.

In modern programming languages, such as *Java 8* and later, *interfaces* have been extended to include *default methods*, which allow methods to have concrete implementations within the *interface*. This feature blurs the line between *abstract* classes and *interfaces* to some extent, but *interfaces* still primarily focus on defining contracts rather than sharing standard functionality, as *abstract* classes do.

Regarding **relationships**, in *UML*, they are used to describe how *classes* are connected and interact with each other within a system. There are many types of relationships, some of which are described in the following.

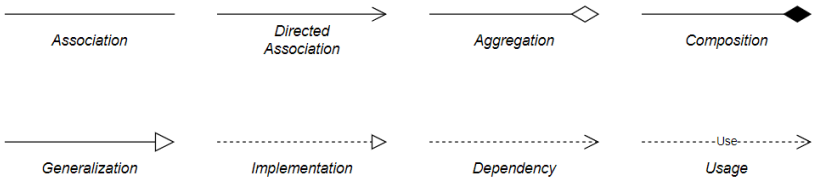


**Figure 2.12:** UML Relationships examples

The **association** is a bi-directional relationship that indicates a generic connection between instances of the connected classes. This type of relationship does not state any surrounding details; it is just used to assert that these entities are somehow related within a system. For instance, a *Book* class and an *Author* class can be linked through an *association* to indicate that a *book* is written by an *author*, without stating anything more.

The **directed association** is a relationship similar to the *association*. Still, it has a direction stating that one class initiates the relationship with the other, which is said to be targeted or affected by the relationship (the targeted class is the one pointed by the arrow). For example, a *Teacher* class can have a *directed association* to a *Course* class to indicate that a *teacher* initiates the relationship by teaching a specific *course*.

The **aggregation** is a relationship representing the *whole-part* concept; this means that if two classes are linked through this relationship, one class contains the other (the container class is the one with the empty diamond shape on its side). Note that the contained class can exist independently of the container class. For example, a *Reunion* class can have an *aggregation* relationship with a *Person* class to state that many *people* can participate in a *reunion*; still, if the *reunion* ceases to be, the *people* can exist independently.

The **composition** is a relationship similar to the **aggregation**, as it states that an instance of a container class is composed of instances of another class (the container class is the one with the filled diamond shape on its side). The critical difference is that the composing class instances cannot exist if the container class instance ceases to exist; for example, a *Digital Book* class can have a *composition* relationship with a *Page* class to indicate that a *digital book* is composed of many *pages*; in this scenario, if the *digital book* is deleted, its *pages* cease to exist as well.

The **generalization**, also known as **inheritance**, is a relationship that represents the *is-a* concept, which means that one class, usually referred to as *child* or *subclass*, inherits the properties and behavior of another class, usually referred to as *parent* or *superclass* (the empty arrowhead points to the parent class). For example, in the context of a restaurant, an *Employee* class can be linked through a *generalization* to many child classes, such as a *Waiter* class, a *Cook* class, and a *Dishwasher* class to indicate that many people working in a restaurant have some common characteristics that are inherited by the *Employee* class. Still, each can also have peculiar ones stated in each subclass.

The **implementation**, also known as **realization**, is a relationship indicating that a class realizes features and operations defined by an interface. In this sense, it should be recalled that, in *OOP*, an interface is an abstract entity that provides signatures of particular methods; consequently, if a class implements that interface, it must implement those methods concretely (the empty arrowhead points to the interface that gets implemented). For example, a *Rectangle* class can be linked to an *Area* interface, which is provided with a *calculateArea* signature of a method, through an *implementation* relationship; this indicates that the *Rectangle* class must provide a concrete implementation for *calculateArea*.

The **dependency** is a relationship indicating that a class depends on another class through a connection that is generally looser than other types of relationships because the class that depends on the other can still exist without it, and the class on which the other depends on does not change its functionalities if this relationship ceases to exist (the dashed arrow starts from the class needing the dependency and points to the class that represents the dependence). For example, a *Person* class can be linked to an *Book* class through a *dependency* relationship to state that a *person* needs a book to read its content, provided the *person* has a *readBook* method; still, if the relationship ceases to exist, the *person* continues to exist and the *book* maintains its characteristics.

The **usage** is a relationship indicating a class that utilizes another class to access certain functionalities, making it more explicit that one class cannot perform specific actions without the assistance of the other class; usually, the class that uses is called *client*, while the one that is used is called *supplier*. The notation is identical to the *dependency* one, but it is enriched with the *use* keyword to underline the client-supplier nature of the relationship. For example, a *Computer* class can be linked to an *Electrical Energy* class through a *usage* relationship to state that a *computer* depends on *electrical energy* to function appropriately; if the relationship ceases to exist, the *computer* cannot provide its functionalities anymore.

### 2.2.2 XML

**XML (Extensible Markup Language)** is a versatile, platform-independent standard for encoding documents in a human-readable and machine-readable format; unlike programming languages, *XML* does not perform computations, as it is a markup language that has been designed with a focus on how data is structured and stored. The main characteristic of this language is that it wraps information in tags; each tag name is enclosed between a < and a >, and the actual data must appear between an opening tag such as <`tagName`> and a closing tag such as < /`tagName`>. It should be noted that these tags are not necessarily predefined but are decided by the author of the *XML* document, just as the structure of the comprehensive document is to be decided.

*XML*'s strength lies in its ability to formally represent complex data structures in a simple text format using a hierarchical structure that allows for nested elements, making it ideal for representing anything from small configuration files to complex datasets (this characteristic is known as being *well-formed*). This flexibility makes *XML* a popular choice for data interchange between systems, often as the foundation for custom markup languages in specialized fields. Additionally, *XML*'s standardization and platform neutrality make it a reliable tool for ensuring interoperability between diverse systems and applications [4].

In the context of the *Security Capability Manager* framework, *XML* has been adopted for specific characteristics that made it suitable for the goal of the project; for instance, its tags, which can be default tags or customized ones, provide explicit context for the data they enclose, making it easier to understand and process the information. Moreover, since *XML* documents are plain text files, they can be created, read, and processed across different tools of the *Security Capability Manager* framework.

It follows an example snippet of how an *XML* file describing a library containing some books might appear:

```
<?xml version="1.0" encoding="UTF-8"?>
    <library>
        <book id="B001">
            <title>XML for Beginners</title>
            <author>John Doe</author>
            <published>2022-05-10</published>
            <price>29.99</price>
        </book>
        <book id="B002">
            <title>Advanced XML</title>
            <author>Jane Smith</author>
            <published>2021-12-01</published>
            <price>35.50</price>
        </book>
    </library>
```

As *XML*'s tags are custom, they can be defined to make the content self-explanatory, which is useful mainly for human stakeholders. In this case, a *library* object is defined, which contains two *book* objects; each *book* has an *id* attribute and some child elements, namely a *title*, an *author*, a *published* date and a *price*. An issue here arises regarding the logical consistency of this *XML* snippet. While it is well-formed thanks to the *XML* grammar being respected if the data enclosed in the *published* tag is changed to a string not representing a date, it would lose its logic while remaining well-formed. To face this issue, *XSD* documents must be introduced.

### 2.2.3 XSD

**XSD (XML Schema Definition)** is a tool used to define and enforce the structure, content, and semantics of *XML* documents. While *XML* provides a way to structure data, it does not inherently guarantee that the data adheres to any specific logical rules beyond basic syntactic correctness. An *XML* document may be well-formed if it follows proper syntax but still contains inconsistencies or incorrect data concerning the specific requirements of an application or system. *XSD* addresses this by providing a mechanism to validate *XML* documents against a predefined schema, ensuring that they are syntactically correct, logically valid, and consistent with the intended data model. In particular, *XSD* defines the constraints for elements and attributes within an *XML* document, such as which elements and attributes can appear in a document, the number and order of child elements, data types for elements and attributes, and eventual default values.

*XSD* follows the same semantic rules of *XML*, so, other than using the same approach regarding tags, it is easily extensible with new regulations and constraints regarding the *XML* documents it refers to. Moreover, it can define simple data types, such as integers or strings, and complex types with multiple nested elements and attributes. Consequently, *XSD* can serve as the blueprint for *XML* documents, defining the rules that the *XML* data must follow and ensuring that *XML* documents are not only well-formed but also valid according to user's logic [5].

In the context of the *Security Capability Manager* framework, *XSD* files are used because of the support they provide for a wide range of built-in data types and for creating custom data types to meet specific needs, regardless of their complexity; since the framework requires non trivial data structures to represent the needed data and achieve its functionalities, validating the *XML* files through *XSD* is a mandatory choice. Moreover, given the mechanism to validate *XML* documents against the defined schema in the *XSD* files, the data that will be described in *XML* files will be automatically checked without the need for human intervention.

It follows an example snippet of how an *XSD* file setting rules and constraints for the library described in the previous section might appear:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string"/>
              <xs:element name="published" type="xs:date"/>
              <xs:element name="price" type="xs:decimal"/>
            </xs:sequence>
            <xs:attribute name="id" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The root element of the *XML* document is a *library* and is a complex type; each library can have an unlimited number of *book* elements, as indicated by the *unbounded* option. The book element is a complex type and contains a sequence of child elements and an attribute; these are a *title*, a string type element; an *author*, a string type element; *published*, a date type element; a *price*, a decimal type element. A *book* also has a string *id* attribute; the *required* option specification makes this attribute mandatory for each book element.

### 2.2.4 XMI

**XMI (XML Metadata Interchange)** is a standardized format used for exchanging metadata information via *XML*; it is primarily used in the context of model-driven engineering, where it serves as a bridge between various modeling tools and environments. *XMI* is particularly important in software development, where it is employed to exchange models created using *UML* and other standards. In particular, *XMI* is used to serialize mainly *UML* models in *XML*, allowing the saving and transfer of systems' components and related relationships between different modeling tools and improving interoperability across various stages of software development. It can be assessed that *XMI* leverages *XML*'s flexibility and extensibility to represent complex metadata, including software models, in a machine-readable and platform-independent format [6].

In the context of the *Security Capability Manager* framework, *XMI* format is adopted to enable the export of the system's *UML* models and other metadata in a standardized way so that it is possible for different tools of the framework to operate on them and execute the functionalities that will be presented in later chapters of this thesis work.

## 2.3 strongSwan

Once the generic structure of *IPsec* is understood, it is clear that the basis on which the protection of a communication channel stands is the definition of this channel, which can be thought of as an abstract concept to indicate the gathering of algorithms and procedures that are applied on a particular data traffic so that it complies with specific security properties; how this is achieved in real scenarios depends on the implementation.

**strongSwan** is an *IPsec* implementation with particular focus on the *IKE* protocol. Its primary purpose is to offer an implementation of the *IKEv2* protocol (*IKEv1* is supported but not recommended) to establish *SA*s between two communicating peers, as well as negotiating *SP*s for them. It must be noted that *strongSwan* does not directly handle *IPsec* traffic; it just installs the *SA*s and the *SP*s that have been negotiated into the operating system's kernel, which will take care of it along with the network itself.

Taking into account its components, the *strongSwan* application responsible for the actual execution of the *IKEv2* protocol is the *charon* daemon; this daemon can be configured and controlled by a command line utility that is known as *swanctl*, through an interface plugin called *vici*. A brief overview of these components will be presented to clarify their logical links. Note that in legacy scenarios, the *swanctl* utility's functions were performed by the *ipsec* utility, which communicated with another daemon called *starter*, which in turn configured the *charon* daemon; these two components will not be considered as they are deprecated [7].

### 2.3.1 charon

To implement the *IKEv2* protocol and establish secure connections among peers, *strongSwan* relies on **charon**, which is a daemon built purposely for this goal. It has many components that are used to manage the *IKE_SA*s and the related *CHILD_SA*s, as well as proper methods to communicate with the device's kernel and install all the *IPsec* associated elements such as *SA*s, *SP*s, etc. If they implement a proper interface, these plugins can register with the daemon at startup and insert themselves into its functionalities. The *vici* plugin interface is among the supported plugins.

### 2.3.2 vici

To configure and control the *charon* daemon, *strongSwan* provides the **vici** plugin, which is one of the many *charon*'s plugin and integrates the *VICI (Versatile IKE Control Interface)* in the project. This interface is specifically suited for *IPC (Inter-Process Communication)* and was designed to face the problem of providing different systems with the necessity to automate the interaction with the *IKE* daemon; the advantage is that external tools can

configure and control the *charon* daemon without the need for human intervention. *VICI* can be seen as an application that implements the server side of this *IPC* protocol that solves the issue above and uses a request/response system with event messages to communicate with the external utilities reliably. In particular, external tools can send requests to configure connections, load certificates and secrets, or retrieve status information. The *vici* plugin responds to these requests and can send notifications or event messages to indicate status changes or important events, such as connection state changes or certificate expirations. This event-based communication is beneficial for real-time monitoring and adjustments in dynamic *IPsec* environments. In this scenario, *swanctl* is the utility that relies on this plugin to communicate with the *charon* daemon and configure its functions.

### 2.3.3 swanctl

As anticipated, **swanctl** is a command line utility that, through the *vici* plugin, can be used to configure, control and monitor the *charon* daemon. This tool is integral to managing *IPsec*-based *VPN (Virtual Private Network)* configurations, allowing administrators to interact dynamically with the *IKE* daemon to manage secure connections. Many sub-commands can be used to instruct the *IKE* daemon about how to manage connections or to view current connection states, retrieve detailed logs, and obtain information about the daemon's overall status. Moreover, there are specific commands that have a `--load-` prefix and which are used to read information, such as connections, secrets, and *IP* address pools, from a particular file, namely the `swanctl.conf` file; this file represents the central point for what concerns the definition of secure channels and it will be thoroughly described later.

### 2.3.4 Configuration directories and files

To manage the settings provided by *strongSwan*, the user is offered a series of configuration directories and files, which can be modified to control the details through which the daemon eventually builds secure connections once executed. For this thesis' purposes, the `strongswan.d` directory and the `strongswan.conf` file will be presented, as these are used for the general configuration of the whole tool and carry much of the syntactical approach that is followed by all the others. Then, the `swanctl.d` directory will be described, as this contains all the files and sub-directories required by the *charon* daemon to implement the *IKEv2* procedure; great focus will be given to the `swanctl.conf` file, which is the main target of this thesis work validation and is the one responsible for defining the security details of the *IKE_SA* and the consequent *CHILD_SA*s.

Information on the legacy *ipsec.conf* file and its related components will not be dispensed as they follow a rigid and non-hierarchical syntax, for which reason they are considered deprecated, and configuring *strongSwan* through them is not recommended anymore; still, support for them is kept for legacy applications.

### 2.3.5 strongswan directory

The content of the **strongswan.d** directory is located by default at `/etc/strongswan.d` and gathers many configuration snippets that can be included in the `strongswan.conf` file by employing instructions resembling `include strongswan.d/file_to_include.conf`. It also contains a `charon` sub-directory that carries configuration snippets for enabled and installed plugins; as an example, since the *vici* interface plugin is installed and enabled for the correct usage of the *swanctl* command tool, it is possible to find a `vici.conf` file that has the following template:

```
vici {

    # Whether to load the plugin. Can also be an integer to increase the
    # priority of this plugin.
    load = yes

    # Socket the vici plugin serves clients.
    # socket = unix://${piddir}/charon.vici
}
```

These configuration snippets for the *charon* daemon are also included by default in the *strongswan.conf* file employing the instruction `include strongswan.d/charon/*.conf` in the *plugins* subsection of the *charon* section.

### 2.3.6 strongswan.conf

Given the incremental nature of the *strongSwan* project, the syntax used, by instance, in the legacy *ipsec.conf* file, due to its specific and hardly scalable format, is not the appropriate choice to describe options for all the related components of *strongSwan*; the **strongswan.conf** file has been introduced from version *5.1.2* as a solution to the necessity of defining general configurations in a way that they can be accessible to extend and readable by all the other *strongSwan* applications.

#### Basic structure

The *strongswan.conf* file consists of a series of hierarchical sections defined by a list of key-value pairs. A section has a name and is followed by its body enclosed by '`{`' and '`}`'; the body can be recursively composed of sub-sections that follow the same semantic rule and key-value pairs in the form of `key = value` (note that each key-value pair has to be terminated by a newline character). A line can be interpreted as a comment if it starts with a '`#`', and the indentation is not mandatory.

An example of a file following the `strongswan.conf` file syntax can be:

```
key1 = value1
#Comment 1
section1 {
  key2 = value2
  subsection1 {
    key3 = value3
  }
  key4 = value4
}
#Comment 2
section2 {
  key5 = value5
}
```

An observation worth adding is that sections and key names are arbitrary and can contain all possible printable characters except for the ones that might cause ambiguities of parsing (’.’ ’,’ ’:’ ’{’ ’}’ ’=’ ’"’ ’#’ ’\n’ ’\t’ ’space’).

### References

One feature introduced with version *5.7.0* is the possibility of referencing other settings and sections to inherit all their key-value pairs with their absolute names; in case of necessity, the value of keys can be overridden by assigning a new value in place, which is helpful since the level of inclusion cannot be currently limited. It is possible to clear the value of an included option by assigning an empty value so that its default value, if defined, will be applied.

In general, a section can apply this inheritance by writing the name of the section, like in the normal case, and then following with a ’:’ and a non-empty list of section names that are consequentially inherited; multiple sections have to be separated by a ’,’. As an example, writing `newSection : refSection1, refSection2 {body_of_newSection}` indicates that `newSection` will inherit all key-value pairs and subsections of `refSection1` and `refSection2` in its body.

The referenced sections are searched starting from the most external scope of the file, which means that if only a subsection should be inherited, it is possible to use a recursive syntax by naming the chain of the sections from the most external to the required one, splitting them with a ’.’ As an example, if a `sub-sub-section` has to be inherited by a `newSection`, this can be obtained by writing `newSection : section.sub-section.sub-sub-section {body_of_newSection}`. Consider that references are resolved at runtime, so it is not an issue if a reference points to a section defined later in the file.

A generic example including all these cases is:

```
conn-defaults {
  # conn-defaults body
}
eap-defaults {
  # eap-defaults body
}
child-defaults {
  # child-defaults body
}
connections {
  conn-a : conn-defaults, eap-defaults {
    # inherits everything from conn-defaults and eap-defaults
    children {
      child : child-defaults {
        # inherits everything from child-defaults
      }
    }
  }
  conn-c : connections.conn-a {
    # inherits everything from conn-a, which means everything
    # from conn-defaults, eap-defaults and child-defaults,
    # since the subsections are also recursively inherited
  }
}
```

### Including files

Another helpful feature is the possibility to include external configuration files in the `strongswan.conf` file; as a consequence, a file with the already discussed semantic can be included in a specific place of the file, and all the settings will be incorporated; this is achievable employing the instruction `include path/of/the/file`, in which `path/of/the/file` can be a relative or an absolute path and might also include the standard shell wildcards. As an example, the name of the file could be referenced by writing `include path/of/the/*.conf`.

The settings that get included are limited to the scope in which the `include` instruction is placed, which can be a section at any level of nesting; the general rule is that the imported sections are added if sections with the same name are not already present; otherwise, they extend the homonym ones by recursively adding subsections and key-value pairs in the original one. Note that if a key-value pair with a homonym key already exists at the same level as the included settings, the current value is overwritten by the imported one.

An example of the inclusion of external files is reported, which, once the inclusions are resolved, actually represents the same configuration file example reported in 2.3.6

**strongswan.conf**

```
key1 = value1
#Comment 1
section1 {
  key2 = beforeinclude
  include include.conf
}
include another.conf
```

**include.conf**

```
# settings included from this file are added to section1
# the following replaces the previous value
key2 = value2
subsection {
  key3 = beforeanother
}
key4 = value4
```

**another.conf**

```
# this extends section1 and subsection
section1 {
  subsection {
    # this replaces the previous value
    key3 = value3
  }
}
section2 {
  key5 = value5
}
```

### 2.3.7 swanctl directory

The content of the **swanctl** directory is located by default at `/etc/swanctl` and contains the `swanctl.conf` file, as well as many sub-directories gathering file-based credentials and private keys used by the *charon* daemon to implement the *IKEv2* protocol.

### 2.3.8 swanctl.conf

With the introduction of the *swanctl* utility to manage the *charon* daemon, a new configuration file has been introduced, namely the **swanctl.conf**, which substitutes the legacy `ipsec.conf` file; this configuration file provides connections, secrets, and *IP* address pools for *swanctl* to control the *charon* daemon. The default location of this file is `/etc/swanctl`, it is defined as a series of key-value pairs and follows the same syntax rules that were presented for the `strongswan.conf` file.

To define these settings and describe how channel protection between devices should be achieved, the file can be edited by changing the values of the supported options, noting that if an option is not explicitly assigned, it automatically gets the default value (provided it has one). The *swanctl.conf* file is characterized by four main sections:

- **connections**: this is the most articulated section; it allows defining the actual *IKE* connection configurations and precisely the details for the *IKE_SA* on which all the *CHILD_SA*s will eventually rely on; each connection contained in the *connections* section is described as a sub-section with a unique name and in which the supported options can be defined. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_connections`.

  In addition, to complete the definition of the *IKE_SA* and to instruct the application about eventual *CHILD_SA*s, three sub-sections appear in this section:

  - **local**: this sub-section allows adding specifications for a local authentication round, which is how authentication is performed for the local peer; for more advanced usage, if multiple rounds have to be instantiated, one round can be defined as a section having the '*local*' keyword as prefix and a unique suffix, which is not requested if only one authentication round is required. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_connections_conn_local`.
  - **remote**: this sub-section allows adding specifications for a remote authentication round, which is how external peers must authenticate to use this connection; for more advanced usage, if multiple rounds have to be instantiated, one round can be defined as a section having the '*remote*' keyword as prefix and a unique suffix, which is not requested if only one authentication round is required. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_connections_conn_remote`.
  - **children**: this sub-section allows adding specifications for a *CHILD_SA* configuration; since each *IKE_SA* may generate more than one *CHILD_SA*, the general rule is that the *children*'s sub-section in each connection of the

*connections* section should have a unique name within the connection itself and the different child configurations can be independently modified. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_connections_conn_children`.

An example of how a *connections* section would appear is presented:

```
connections {
    connectionName1 {
        local_addrs  = value1
        local {
            auth = value2
            id = value3
        }
        remote {
            auth = value4
        }
        children {
            childName1 {
                local_ts = value5
                updown = value6
                esp_proposals = value7
            }
        }
        version = value8
        proposals = value9
    }
    connectionName2 {
        local_addrs  = value10
        local {
            auth = value11
            certs = value12
            id = value13
        }
        remote {
            auth = value14
        }
        children {
            childName2 {
                local_ts = value15
                updown = value16
                esp_proposals = value17
            }
        }
        version = value18
        proposals = value19
    }
}

```

In this example, *connectionName1* and *connectionName2* are arbitrary names and must be unique in the scope of the *connections* section; also *childName1* and *childName2* are arbitrary names for the sub-sections of the *CHILD_SA*s, and they should be unique only in the scope of each single *children* section.

- **authorities**: this section allows defining additional attributes for the certification authorities; each authority contained in the *authorities* section is described as a sub-section in which the supported options can be defined. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_authorities`.

  An example of how a *authorities* section would appear is presented:

```
1    authorities {
2        certificationAuthorityName {
3            cacert = value1
4            ocsp_uris = value2
5        }
6    }
7
```

  In this example, *certificationAuthorityName* is an arbitrary unique name.

- **secrets**: this section allows defining secrets for authentication and private key decryption; it is composed of different subsections having a specific prefix, which represents the secret type and for which some options can be defined. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_secrets`:

  An example of how a *secrets* section would appear is presented:

```
1    secrets {
2        eap1 {
3            id = value1
4            secret = value2
5        }
6        eap2 {
7            id = value3
8            secret = value4
9        }
10   }
11
```

  In this example, *eap1* and *eap2* are names composed of a prefix (*eap*) and a suffix; the suffixes are arbitrary and unique, while the prefixes are fixed keywords.

- **pools**: this section allows defining named pools that may be referenced by connections with the `pools` option, and that can be used to assign virtual IPs and other configuration attributes; similarly, in this case, the name of the pools should be unique. All the possible key-value pairs and their usage are described in the official documentation at `https://docs.strongswan.org/docs/5.9/swanctl/swanctlConf.html#_pools`.

  An example of how a *pools* section would appear is presented:

```
    pools {
        pool1 {
            addrs = value1
        }
        pool2 {
            addrs = value2
        }
    }

```

  In this example, *pool1* and *pool2* are arbitrary unique names.

# Chapter 3

# Related Works

This chapter is dedicated to how past works and results obtained from previous researchers have influenced and guided this thesis. The first part will focus on the research documented in scientific papers, mainly on the paradigms studied to address recent networking issues. The second part will shift to presenting the framework on which this thesis has built its foundations and eventually enhanced, with attention to its general workflow and characteristics.

## 3.1 Context of previous works

As modern networks grow in size, complexity, and demand, traditional networking architectures have proven inflexible, costly, and challenging to scale. The increasing complexity of security management and the rapid growth of sophisticated cyberattacks further complicates the scenarios. To address these limitations, *Network Function Virtualization* and *Software Defined Networking* paradigms have emerged.

### 3.1.1 Network Functions Virtualization

**NFV (Network Function Virtualization)** transforms how network services are deployed and managed by decoupling network functions from the hardware on which they are executed. Traditionally, security services were tied to specific hardware devices, requiring significant capital investment, lengthy deployment times, and costly maintenance efforts. *NFV* shifts these functions into software that can run on commodity hardware, allowing network operators to deploy and update network services as needed without the constraints of physical hardware.

In recent years, the idea of exploring this approach through *NFV* has increased; for instance, reliance on vendor-specific hardware is reduced as *NFV* enables general-purpose servers to run network functions. Moreover, *NFV* makes it easier to scale services based on network demand. Despite this, there is still the issue of standardization; many security services provided in virtualized environments rely on proprietary solutions, which can lead to vendor lock-in and interoperability issues. To solve this, the *IETF (Internet Engineering Task Force)* has been working on developing standard interfaces for *NFV*-based security services, referred to as **NSF (Network Security Function)s** [8].

*NSF* is a term used to underline a function that implements some of the most common security properties; many different *NSF*s can be used to guarantee a customized range of security properties and can be provided by various vendors for different technologies, so the goal was to standardize the interface through which these *NSF*s can be configured in a way that is agnostic of vendors and technologies. The goal is to create standardized interfaces that allow third-party security vendors to integrate easily into virtualized environments.

### 3.1.2 Software-Defined Networking

**SDN (Software Defined Networking)** addresses the separation of the network's control plane from the data plane. In traditional network architectures, each network device has its control plane, making it difficult to manage large-scale networks and apply consistent policies across all devices. This leads to inefficient network management, slow service deployment, and the inability to dynamically respond to changing traffic patterns [9].

*SDN* was one of the paradigms followed by *Basile et al.* [10] for a study that is part of the primary foundation for this thesis work. This is because it introduces a centralized control plane decoupled from the hardware, allowing network administrators to control the network through a centralized controller. This controller has a global view of the network, enabling the implementation of policies across all network devices from a single point of control. Furthermore, *SDN* controls network behavior using software applications, enabling faster service deployment and more dynamic traffic management. Finally, by abstracting network control from the hardware, *SDN* allows networks to adapt to real-time traffic conditions, security threats, or business needs.

## 3.2 Security Capability Manager

This thesis has started from an already existing framework, called **Security Capability Manager**, that proposes a solution to the issue of deploying security functions in different environments with peculiar low-level characteristics; the basic idea has been to develop an environment in which network administrators could describe at a high level the security details they wish to implement for a system, employing a common abstract language.

Then, through proper automatic translation tools, the abstract language would be autonomously translated into low-level specifications that depend on the underlying technologies without requiring the administrators to be informed about them. Given the scalable nature of this framework, vendors can add support for new security applications so that high-level policies can be described and translated into low-level ones.

### 3.2.1 Framework Overview

The framework was realized using a methodology similar to the one adopted for the current thesis work to implement the objectives described in the previous paragraph. The *Modelio* tool was employed to model the architecture of the two main models describing the framework, namely the *Capability Information Model* and the *Capability Data Model*. Then, the processing of these models and the consequent translation of high-level policies to low-level ones is carried out using three *Java* tools, namely the *XMI to XSD Converter*, the *Abstract Language Generator* and the *NSF Translator*. The artifacts required by these tools to properly function are the *XMI Model*, *XSD Capability Data Model*, the *NSF Catalogue*, the *XSD Grammar* for a specific *NSF*, an *XML Rule* for a particular *NSF*, and a *Low-Level Policy* with the translated policy. These main components will now be described, as possessing explicit knowledge of their roles in the framework is crucial to understanding the implementation of this thesis work.

### 3.2.2 The models

This section will present the models that comprise the framework's design, underlining their purpose theoretically and in this thesis's context.

#### Capability Information Model

An *Information Model* typically focuses on concepts and their relationships, independent of any specific implementation or technology, and defines the meaning and rules of the data without worrying about how that data is stored or physically represented; technical details are consequentially left for other models.

The **CapIM** is a model that describes the framework and its functionalities in a general way. In the context of the *Security Capability Manager*, the *CapIM* describes *NSF*s abstractly and their relations with *Security Capabilities*. In modeling these entities, the focus is provided to how the different *NSF*s and the related *Security Capabilities* have to be translated into low-level specifications, employing a class known as *NSFCatalogue*, which is linked to classes called *CapabilityTranslationDetails* and *NSFPolicyDetails*; these will play a significant role in the implementation of channel protection.

#### Capability Data Model

A *Data Model* is a more concrete and implementation-specific representation of how data is stored, structured, and manipulated within a software system; it directly reflects the underlying data structures, such as databases, file systems, or data formats, and includes details about how data is managed. The **CapDM** is a model that describes the system and its functionalities in a more specific way, which is suitable for developers who need to get knowledge of technical details.

This artifact is much larger and more complex than the *CapIM*, as it provides specifications for all the actual *Security Capabilities* supported by the framework. Moreover, the *CapDM* partitions the capabilities according to their logical structure; in particular, some partitions are *Condition Capabilities*, in case they represent a condition to be checked according to specific condition operators, and *Action Capabilities*, in case they represent a security action to be performed.

### 3.2.3 The Artifacts

This section presents the artifacts that the Java tools must utilize to execute their functionalities, specifying each artifact's purpose and structure.

#### XMI Model

The **XMI model** contains the *UML* model developed for this thesis, alongside all model components implemented in previous work. This file must be provided as the initial element of the workflow so that the system's design is available in a machine-readable format. This allows the framework to process the design and initially derive the grammar used to validate the *NSF catalogue*, followed by the grammar required for translating specific *NSF* rules.

#### NSF Catalogue

The **NSF catalogue** contains all the information about the implemented *NSFs* within the framework. Specifically, it provides detailed instructions for translating each *NSF*, as well as descriptions of the *security capabilities* it supports, including how each capability should be translated when encountered in the *XML Rule* file. Although examples are provided here, complete documentation on all the elements and possibilities in this artifact can be accessed in the framework's implementation documentation.

For instance, details about a specific *NSF* can be defined as follows:

```
<nSF id="StrongSwan">
    <nsfPolicyDetails>
        <ruleStart>ike-name-placeholder {\n\t</ruleStart>
        <ruleEnd>\n}\n</ruleEnd>
        <policyTrailer/>
        <policyEncoding/>
        <capabilityGroup>
            <capabilityGroupName>local</capabilityGroupName>
            ...
            <securityCapability ref="SourceAuthActionCapability"/>
            <securityCapability ref="AaaIdentityConditionCapablity"/>
            ...
        </capabilityGroup>
        ...
    </nsfPolicyDetails>
    <securityCapability ref="AaaIdentityConditionCapability"/>
    <securityCapability ref="AETechniqueActionCapability"/>
    ...
</nSF>
```

An *NSF* always includes an `nsfPolicyDetails` element, which contains `ruleStart` and `ruleEnd` elements to specify strings that are appended at the beginning and end of each rule, respectively. It also contains `policyTrailer` and `policyEncoding` elements, which define a string to be appended at the end of the entire policy and specify the encoding used when printing the translated policy.

Additionally, a variable and optional number of `capabilityGroup` elements are discussed in more detail in another chapter. The `nSF` element also contains multiple `securityCapability` elements that list all *security capabilities* implemented by the *NSF*.

Each *security capability* must include details about how it should be translated for a specific *NSF*, since multiple *NSFs* can support the same `securityCapability` but may define it differently at the implementation level.

Here is an example of a `capabilityTranslationDetails` element:

```
<capabilityTranslationDetails>
    <nSF ref="StrongSwan"/>
    <securityCapability ref="AaaIdentityConditionCapability"/>
    <commandName>
        <realCommandName>aaa_id</realCommandName>
    </commandName>
    <internalClauseConcatenator>=</internalClauseConcatenator> <
    clauseConcatenator>\n\t</clauseConcatenator>
</capabilityTranslationDetails>
```

In this case, the element contains an `nSF` to indicate which *NSF* the translation details refer to, a `securityCapability` element to specify the name of the capability (which must be included in the previously defined list), a `commandName` element to specify the low-level command for the current *NSF*, and finally an `internalClauseConcatenator` and a `clauseConcatenator` to indicate, respectively, a string that separates the command name from the capability value, and a string appended at the end of the capability.

### XSD Capability Data Model

The **XSD Capability Data Model** contains the grammar for validating the *NSF catalogue*. The framework needs this artifact to ensure the logical correctness of the catalogue, allowing it to generate the correct grammar for each *NSF*, which will be used for translating the *XML Rule*.

### XSD Grammar for an NSF

The **XSD Grammar for an NSF** is an artifact generated to validate a high-level rule provided to the framework for translation. Before translation, the framework ensures the provided rule complies with the logic described in this file. Below is an example of how this type of grammar is structured, referencing the same capability used in the previous example:

```
<xs:complexType name="IdentityCapability">
    <xs:complexContent>
        <xs:extension base="ConditionCapability">
            <xs:choice maxOccurs="unbounded" minOccurs="0">
                <xs:element maxOccurs="1" minOccurs="0" name="identity"
   type="xs:string"/>
            </xs:choice>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="AaaIdentityCapability">
    <xs:complexContent>
        <xs:extension base="IdentityCapability"/>
    </xs:complexContent>
</xs:complexType>
<xs:complexType name="AaaIdentityConditionCapablity">
    <xs:complexContent>
        <xs:extension base="AaaIdentityCapability"/>
    </xs:complexContent>
</xs:complexType>
```

A `complexType` element is defined with the name `AaaIdentityConditionCapability` to specify the *security capability* this grammar is concerned with. Within the element's body, it states that this type extends another capability using the `extension` element. Specifically, it extends the `AaaIdentityCapability`, which, in turn, extends `IdentityCapability`, indicating that `AaaIdentityCapability` is part of a broader group of *condition capabilities*. The `choice` element further specifies that this capability can optionally occur at most once and must contain a string value enclosed in `identity` tags.

### XML Rule

The **XML Rule** contains the abstract rule to be translated by the framework according to the constraints defined in the previous files. Below is an example of how an abstract capability can be structured:

```
<rule id="0">
    ...
    <AaaIdentityConditionCapability>
        <identity>someIdentity</identity>
    </AaaIdentityConditionCapability>
    ...
</rule>
```

In this case, the `AaaIdentityConditionCapability` is an abstract capability that contains the attribute `identity`, as specified in the corresponding *XSD*. This attribute contains the value `someIdentity`, which represents the name of the authentication server used during authentication.

### Low-Level Policy

The **Low-Level Policy** is the output of the framework's workflow, representing the policy translated into the specific *NSF*'s language. Based on all the examples provided above, the following snippet illustrates how the final translation may appear:

```
...
aaa_id=someIdentity
...
```

In conclusion, the `AaaIdentityConditionCapability` has been translated into *strongSwan*'s specific syntax, demonstrating how the framework transforms abstract rules into a format compatible with the target *NSF*.

### 3.2.4 The Java Tools

This section will present the Java tools that carry out the framework's functionalities, focusing on the execution flow linking each to the already discussed artifacts.

#### XMI to XSD Converter

When the system's model is complete, it must be converted into the *XMI* format to make it processable by the framework; in this sense, *Modelio* provides the possibility to export models automatically into *XMI*. Once this artifact is obtained from *Modelio*, the *Security Capability Manager* framework is capable of generating the related *XSD* file; this file describes to which semantic and logical restrictions the *NSF Catalogue*, which is an *XML* file, must comply. To obtain the *XSD* artifact from the *XMI* artifact, it is necessary to execute the *Converter* Java class `main` method giving the *XMI* file as input; the output is the *XSD* file.

#### Abstract Language Generator

Once the *XSD* file for the *NSF Catalogue* is obtained, it is necessary to generate the *XSD* file that provides syntactic and semantic restrictions for *Rule Instance* artifacts of specific *NSF*s; this means that since the final goal is to provide the framework with an *XML* file containing a high-level rule and receive as output some *NSF* specific low-level rule, the translation process will require a way to validate the rule mentioned above concerning the technical details of the target *NSF*. As a consequence, the *Security Capability Manager* framework provides a *LanguageModelGenerator* class with a `main` method; by providing this tool with the *NSF Catalogue* file and its related *XSD* file, and specifying a target *NSF* among those supported by the framework, this tool will generate as output the *XSD* artifact for the desired *NSF*, so that translations towards that *NSF* can be executed by the following Java tool.

#### NSF Translator

The **Translator** Java class is the last tool of this translation execution flow. Executing the `main` method of this class and providing as inputs the *XML* file containing the *Rule instance*, as well as the *XSD* for the target *NSF* and the *NSF Catalogue*, will result in the generation of the low-level policy for the requested *NSF*. As previously anticipated, this tool uses the *XSD* input file to check the syntax and the semantics of the *Rule Instance*. In contrast, the *NSF Catalogue* provides all the necessary translation details for the capabilities contained in the rule.

# Chapter 4

# Solution Design

Once the surrounding topics have been clarified, it is appropriate to shift focus to how the primary goal of this thesis was approached. All modifications and contributions to the existing framework were built upon previous work, particularly the results of *Cirella*'s thesis, which provided a valuable starting point for building the solution for this work [1]. These contributions were carried out following the techniques and best practices discussed earlier. The expected outcome of the design phase was to have a formal and rigorous model describing what characterizes channel protection so that security properties, regardless of the protocols and algorithms used by particular implementations, can be identified and described in the most versatile way possible. The model has been realized through *UML* and primarily using the functionalities provided by *Modelio*, the same modeling tool adopted for previous work.

## 4.1 Problem Statement

Communication channels are among cybersecurity's most critical security systems, as malicious users can exploit them to target weaknesses and vulnerabilities. These channels serve as the medium for data transmission between systems, making them prime targets for attacks such as data interception, manipulation, and unauthorized access. However, as discussed in previous sections, navigating the technologies that implement channel protection protocols and procedures can be challenging, even for domain experts, due to the growing complexity and variety of cyber threats and the sophistication of attackers. The goal is to provide an abstraction layer for channel protection details, allowing a high-level representation of security mechanisms and making it easier to describe the security capabilities of *NSF*s for secure channel communication. Much focus will be on expanding the abstract language to cover the necessary aspects of channel security and enabling seamless interaction with all channel protection *NSF*s.

## 4.2 Use Cases

Exploring new ways to address vulnerabilities in communication channel security is essential to protecting sensitive information from unauthorized access, interception, and manipulation. Since attackers frequently exploit weaknesses that provide access to critical data, insecure communication channels can expose systems to external threats and lead to human-induced errors in their configuration, further increasing the likelihood of security breaches.

Reducing human-induced errors in channel configurations can be beneficial in a wide range of real-world scenarios. Vulnerabilities can manifest themselves in different ways, depending on the nature of the communication channel and the type of data being transmitted. Compromised communication channels can result in both financial and reputational losses.

Outlined below are some scenarios where the solution provided by this framework could mitigate related security risks:

- *Data Interception*: One of the most common communication channel threats occurs when attackers overcome encryption techniques and eavesdrop on data transmissions. Consequently, exploiting weak encryption or unsecured channels can steal sensitive information such as login credentials, personal data, or proprietary business information.

- *Data Integrity*: Ensuring data remains unchanged during transmission is one of the most critical security properties in scenarios this thesis work addresses. Insecure communication channels may allow attackers to alter messages in transit, potentially leading to critical consequences.

- *Authentication and Authorization*: Secure communication channels must support algorithms that ensure the entities involved are who they claim to be. Weak channel security allows attackers to spoof their identities and gain unauthorized access to sensitive systems.

- *Malicious Software Propagation*: Vulnerabilities in communication channels can be exploited to make them mediums for distributing malicious software, such as ransomware or spyware.

- *Compliance with Regulations*: Many industries are subject to national and international data protection regulations, such as the already mentioned *GDPR*. Ensuring secure communication channels is often a legal requirement, and failing to protect data in transit can result in legal penalties and fees other than the consequent loss of reputational value.

## 4.3 Requirements for the design phase

The realization of the model has undergone different realization and refinement steps, as will be pointed out in later paragraphs; still, best efforts have been made in continuously following these general requirements:

- *Abstraction Layer*: The model must introduce a transparent abstraction layer that can generalize communication channel security properties, enabling a versatile description of channel security and allowing the integration of different technologies while maintaining a consistent and unified representation of security mechanisms.

- *Modularity and Extensibility*: The model should be designed modularly to add or update different security mechanisms without affecting the entire system. It should support easy expansion for future developments or updates in channel protection techniques.

- *Separation of Concerns*: Different aspects of communication channel security should be separated within the model to ensure clarity and avoid conflicts in distinct security principles. This approach should allow specific issues to be addressed independently, simplifying maintenance and updates.

- *Consistency with Previous Work*: The model should build upon existing work, particularly the contributions of *Cirella*'s thesis, ensuring that any modifications or extensions respect the architectural decisions and principles established in previous efforts. It should integrate seamlessly with prior models, providing continuity in the research.

- *Security Capability Representation*: The model must adequately represent the security capabilities of *NSF*s related to secure communication channels. It should be able to define how various *NSF*s handle encryption, authentication, data integrity, and other security mechanisms involved in channel protection.

- *Protocol and Implementation Agnostic*: The design should abstract from specific protocol implementations, allowing the model to describe security properties independently of particular technologies. This approach would ensure the model remains relevant even as protocols evolve or change, emphasizing security concepts rather than technical specifics.

- *Support for Real-world Use Cases*: The model should be capable of describing use cases that reflect real-world scenarios, such as preventing data interception, ensuring message integrity, and protecting against unauthorized access. These use cases should guide the design to ensure the model addresses practical concerns and vulnerabilities found in communication channels.

## 4.4 Execution of the design phase

As previously mentioned, the model development required by the new framework's functionalities was carried out in the *Modelio* environment. It involved several iterations to achieve the most suitable design for this thesis. The approach involved creating multiple drafts of a model that could represent channel protection characteristics in an implementation-agnostic manner, following the principles and objectives outlined earlier.

Each draft was followed by high-level validation, during which weaknesses were identified and improvements made to better represent fundamental security properties. Different modeling solutions were explored in these iterations to ensure the model could accommodate diverse security scenarios. In the following paragraphs, the components of the final model will be presented and described, highlighting the reasons behind the chosen solution and how it satisfies the requirements of a formal channel protection model.

### 4.4.1 Base Model Section

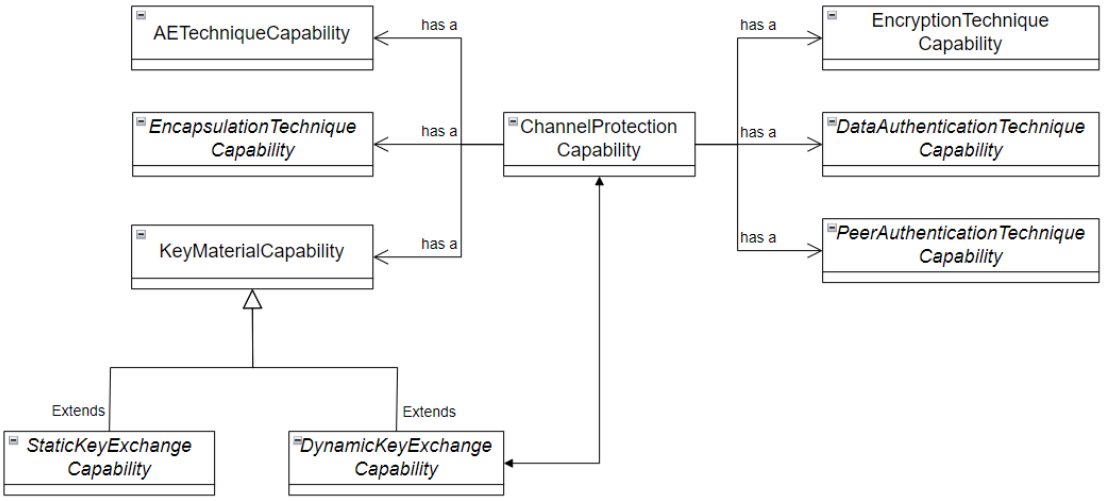


**Figure 4.1:** Base Model

This first part of the model introduces the most generic elements required to secure a communication channel. Regardless of the protocols implemented, a protected channel consistently exhibits the following properties:

- An **encapsulation** technique, defining how data in transit is enveloped before transmission.
- A **data encryption** technique, specifying how data in transit is protected to ensure *confidentiality*.

- A **data authentication** technique, detailing how data in transit is protected to ensure *authentication* and *integrity*, usually by calculating a *MAC (Message Authentication Code)*.
- A **peer authentication** technique, defining how the communicating parties authenticate to each other as proof of identity.
- An **authenticated encryption** technique specifying how data in transit is protected using algorithms and modes that provide *confidentiality*, *data authentication*, and *integrity* together.
- A **key material** element, defining the algorithms and techniques through which the communicating parties exchange the secrets and parameters for channel protection as described in previous points. This exchange can be achieved through static or dynamic methods.

This overview provides a clear understanding of the core components involved in securing a communication channel. It should be noted that the *KeyMaterialCapability* element generalizes two distinct approaches for exchanging the algorithms and parameters required for security, which are the *static* and the *dynamic* approach. In the case of dynamic key exchange, the `DynamicKeyExchangeCapability` element is directly connected to the `ChannelProtection` element, as the exchange occurs over the same channel. In contrast, static key exchange is not directly related to the `ChannelProtection` element as it often takes place *OOB (Out-Of-Band)*.

### 4.4.2 Encapsulation Model Section

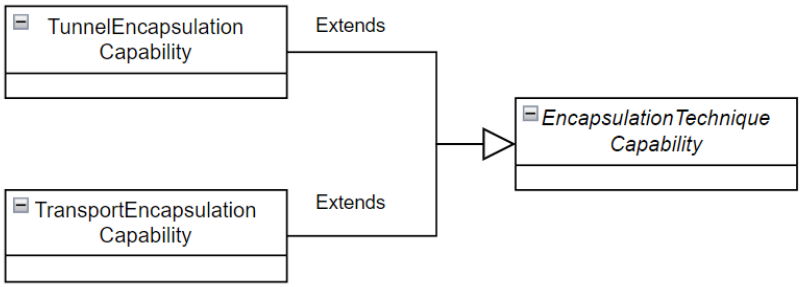


**Figure 4.2:** Encapsulation Model

This part of the model addresses the type of encapsulation data packets undergo before transmission through the network. Specifically, it distinguishes whether a packet is encapsulated using *tunnel* mode or *transport* mode, depending on the communication type it is desired to establish, as illustrated in the *Background* section. It is important to note that both modes are considered implementations of the more general and abstract *EncapsulationTechniqueCapability* element.

### 4.4.3 Data Encryption Model Section

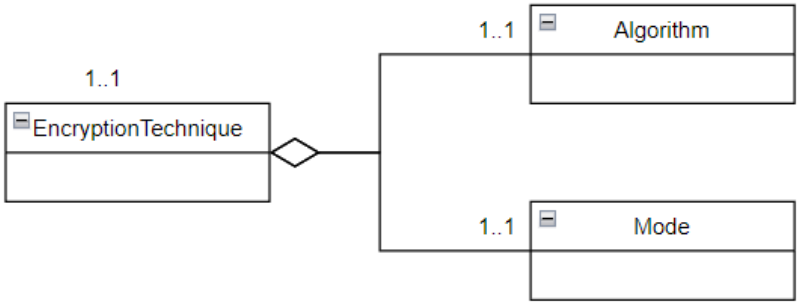


**Figure 4.3:** Data Encryption Model

This part of the model focuses on how encryption is integrated into channel protection, specifying that any generic encryption technique must include:

- an **algorithm** used for the actual encryption procedure, such as *DES (Data Encryption Standard)* or *AES (Advanced Encryption Standard)*.
- an operational **mode** with which the algorithm is applied, such as *CBC (Cipher Block Chaining)* or *CFB (Cipher Feedback)*.

In this case, the generic `EncryptionTechnique` element is designed to be a composition.

### 4.4.4 Data Authentication Model Section

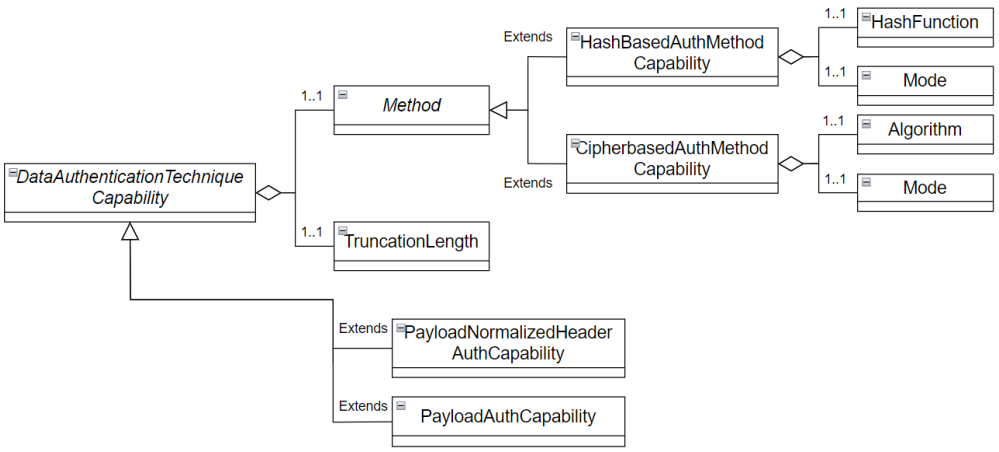


**Figure 4.4:** Data Authentication Model

This part of the model addresses the requirements for ensuring that data received through the channel is intact and authentic. More specifically, a data authentication algorithm requires a method that typically falls into one of two categories:

- **Hash-based**: This method requires a hash function, such as *SHA256*, and an operational mode, typically *HMAC (Hash-based Message Authentication Code)*.
- **Cipher-based**: This method requires a cryptographic algorithm, such as *AES*, and an operational mode, such as *CBC*.

If necessary, the *truncation length* of the *MAC* can also be specified, depending on the scenario (for instance, when using *SHA384*). These elements are implemented as compositions, similar to the other security properties.

Additionally, it is essential to note that data security can be applied to either the packet payload or the header. As a result, the generic `DataAuthenticationTechniqueCapability` can be implemented by either the `PayloadAndNormalizedHeaderAuthCapability` or the `PayloadAuthCapability`.

### 4.4.5 Peer Authentication Model Section

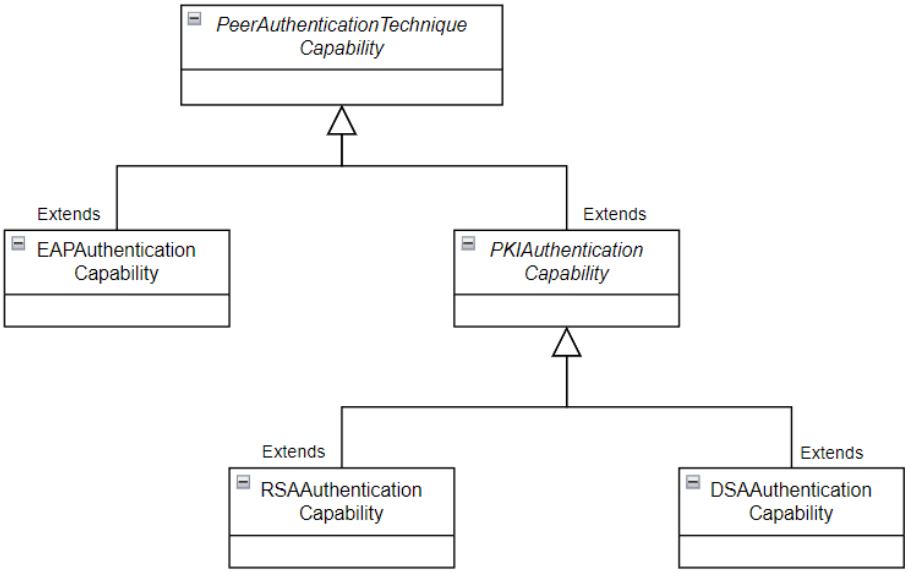


**Figure 4.5:** Peer Authentication Model

This part of the model outlines the methodologies used to verify the identity of parties in secure communication. The primary approaches considered are:

- **EAP-based authentication**, which utilizes the *EAP (Extensible Authentication Protocol)* framework's functionalities to support various authentication methods depending on the network's needs.
- **PKI-based authentication**, which relies on techniques involving *certificates* and *PKI (Public-Key Infrastructure)s*, which provides a higher level of security by using public and private key pairs to authenticate identities. In this case, the `PKIAuthenticationCapability` element is implemented by either `RSAAuthenticationCapability` or `DSAAuthenticationCapability`, depending on the specific requirements of the scenario.

### 4.4.6 Authenticated Encryption Model Section

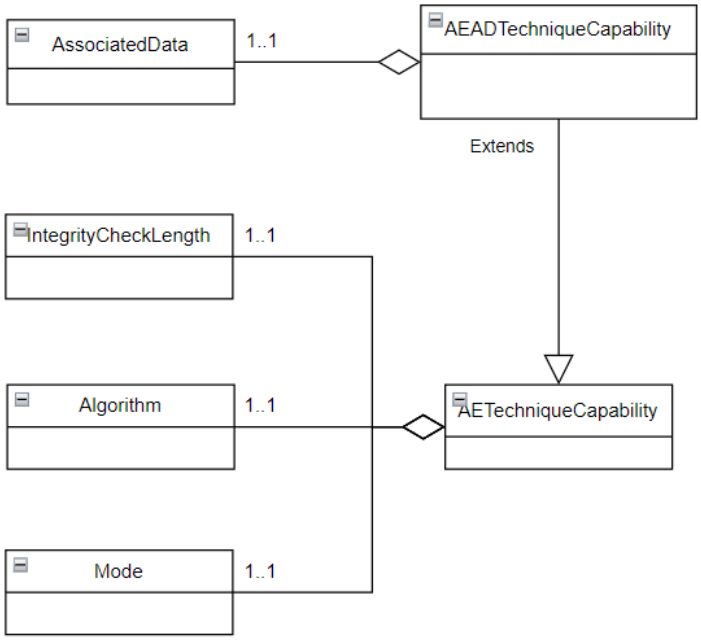


**Figure 4.6:** Authenticated Encryption Model

This part of the model focuses on approaches based on the *AE (Authenticated Encryption)* procedure, which usually requires the following elements:

- An **algorithm** used to achieve these security properties, such as *AES*.
- An operational **mode** for the algorithm, such as *GCM (Galois Counter Mode)* or *CTR (Counter)*.
- An **integrity check length**, which specifies the length of the *MAC* (also referred to as the *authentication tag* in the context of *AE*).

Combining these elements ensures that the data remains confidential during transmission while also providing a mechanism to verify its authenticity and integrity. If required by the usage scenario, *AE* techniques can be extended with additional data that is authenticated but not encrypted, known as **associated data**. When this additional parameter is involved, the process is referred to as *AEAD (Authenticated Encryption with Associated Data)*.

All these elements represent the *AE* technique through the composition approach. Furthermore, as *AEAD* is a specific case of *AE*, the `AEADTechniqueCapability` is implemented as an extension of the `AETechniqueCapability`, ensuring that it inherits all the base functionalities.

### 4.4.7 Static Key Exchange Model Section

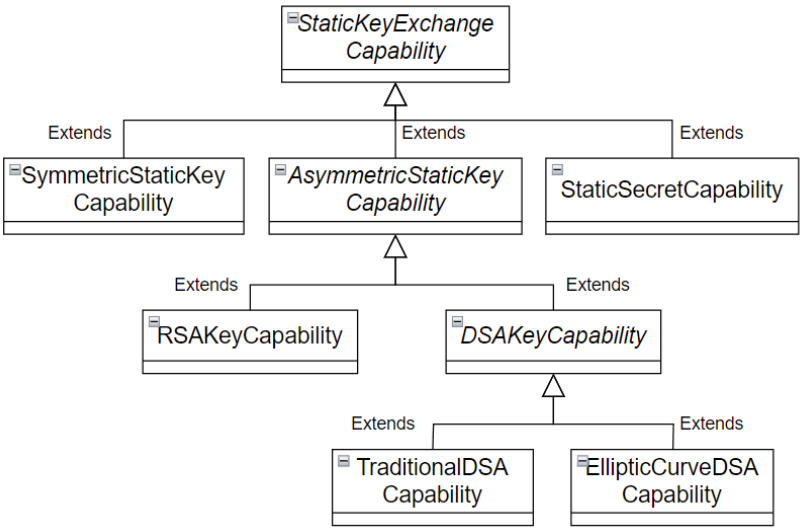


**Figure 4.7:** Static Key Exchange Model

This part of the model examines exchanging key material through static approaches. In this context, the exchange is said to occur *OOB*, meaning that the communication channel is not used to exchange the necessary cryptographic parameters. Instead, these parameters are distributed through alternative methods, including physical exchange or separate communication channels. The parameters involved in this procedure include:

- A **symmetric key**, if the algorithms used during data exchange rely on symmetric encryption approaches.
- An **asymmetric key**, if the algorithms that will be used during data exchange rely on asymmetric cryptographic schemes.

- A generic **static secret**, used in methodologies where the cryptographic key is not immediately involved. Instead, this secret might be used to derive the key or other cryptographic material in a subsequent phase.

In the *asymmetric static key* case, the key is usually involved in performing algorithms such as *RSA (Rivest-Shamir-Adleman)* or *DSA (Digital Signature Authentication)*, which rely on traditional mathematical principles or more advanced methods like *elliptic curves*.

From a modeling perspective, these elements are represented at various levels of implementation, starting with the `StaticKeyExchangeCapability` element and going to more specific use cases.

### 4.4.8 Dynamic Key Exchange Model Section

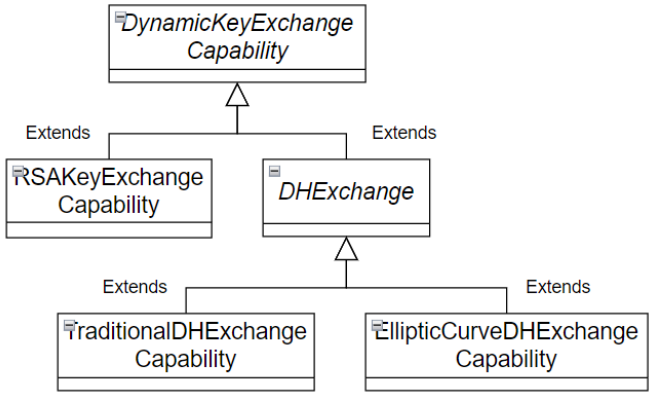


**Figure 4.8:** Dynamic Key Exchange Model

This final part of the model outlines the process of exchanging the key material through dynamic approaches. Unlike the previously described static scenario, the communication channel is used initially to exchange the parameters required to secure the data before it is transmitted. Two primary dynamic methods are considered for this purpose:

- **RSA** key exchange, where the *RSA* protocol is employed to generate asymmetric keys for securing communication.
- **Diffie-Hellman** key exchange, where the *DH* technique is used to establish asymmetric keys.

In the case of *DH*, either the traditional mathematical method or the more modern *ECDH (Elliptic Curve Diffie-Hellman)* approach can be used. From a modeling perspective, these exchange techniques are presented as specific implementations of the `DynamicKeyExchangeCapability` element.

# Chapter 5

# Solution Implementation

This chapter is dedicated to presenting how the solution designed in the previous chapter has been concretely implemented in the framework. By leveraging the results of earlier work and the outputs from the design phase, the implementation process can be broken down into four main steps:

- **strongSwan Integration**: This section explains how support for *strongSwan* has been incorporated into the framework, following the theoretical foundations presented in the related section of the *Background*. Attention will also be given to the challenges encountered during the integration process and the necessary steps to address these issues for complete support.
- **Groups Management**: The need for managing groups of capabilities directly resulted from *strongSwan* integration. This section will explore why this step became mandatory and describe how group management was implemented.
- **POSET Processing for Rule Translation**: This section will explain how the structure known as *POSET (Partially Ordered Set)* (further details to be provided later) was processed to prepare it for the actual rule translation phase.
- **POSET Translation**: The final section focuses on how the refined *POSET* structure from the previous step has been integrated into the main translation workflow, resulting in the generation of low-level rules, which marks the completion of the implementation phase.

Many of these implementation steps were carried out in parallel with the thesis work of *Marchitelli* [11]. Specific elements of the implementation process relied on his contributions, which will be referenced and discussed as necessary.

## 5.1 strongSwan Integration

With the new model for channel protection integrated into the main framework, it became necessary to bring *strongSwan* into scope. To properly validate the framework in the final stages of this thesis, support for a real-world *NSF* focused on channel communication was essential, at least for the most common and relevant scenarios.

In the legacy framework, *strongSwan* already had partial functionality implemented, with support for rule translation available. However, significant rework was required to adapt the structure and security capabilities, primarily because the previous *NSF Catalogue* provided translation details for the *ipsec* plugin of *Background*. As discussed in the *Background*, this plugin required the configuration file to be structured differently from the newer *swanctl* plugin, which is now recommended.

As an example, a basic *IPsec* setup can be assumed with the following requirements:

1. *Local information*: the local peer that initiates the communication channel has *IP* address `192.168.1.1` and belongs to a network with *IP* range `192.168.1.0/24`.
2. *Remote information*: the remote peer has *IP* address `192.168.2.1` and belongs to a network with *IP* range `192.168.2.0/24`.
3. *Parameters dynamic exchange*: performed using `AES128`, `SHA256` and elliptic curve `x25519`.
4. *Peer authentication*: performed by both peers using a *PSK*.
5. *Data authentication and confidentiality*: performed using `AES128` in `GCM128` mode and elliptic curve `x25519`.
6. *Additional Information*: the connection must be initiated immediately without waiting for related traffic to be detected; moreover, the *IKEv2* protocol must be executed.

The *ipsec.conf* file generated from the framework would be as follows (assume that a high-level rule file has been provided, with specific syntax that will be discussed later):

```
    conn examplevpn
    left=192.168.1.1
    right=192.168.2.1
    leftsubnet=192.168.1.0/24
    rightsubnet=192.168.2.0/24
    authby=secret
    ike=aes128-sha256-x25519
    esp=aes128gcm128-x25519
    keyexchange=ikev2
    auto=start
```

In the context of the new plugin, the `swanctl.conf` file would carry a configuration for the same setup as follows:

```
    examplevpn {
        version = 2
        local_addrs = 192.168.1.1
        remote_addrs = 192.168.2.1

        local {
            auth = psk
            id = 192.168.1.1
        }

        remote {
            auth = psk
            id = 192.168.2.1
        }

        children {
            net {
                local_ts = 192.168.1.0/24
                remote_ts = 192.168.2.0/24
                esp_proposals = aes128gcm128-x25519
            }
        }

        proposals = aes128-sha256-x25519
    }
```

It is evident that while many options maintain a direct correspondence, the overall structure has undergone significant changes. Some pre-existing capabilities required only minimal adjustments, while others needed more substantial revisions. The following paragraphs will summarize the most relevant updates to establish a stable view of how the catalogue now supports *strongSwan*'s capabilities in the new configuration version. Specifically, the focus will be on the options that require significant changes and have also been implemented in the framework. In contrast, the ones left for future works will be referenced only in the related *Appendix*.

### 5.1.1 Rekey Parameters Management

In actual *IPsec* configurations, the secure channel is used, as described in the *Background* section, to negotiate multiple *CHILD_SA*s. These connections are typically intended for limited use, as keeping them alive for extended periods increases the risk of becoming vulnerable to cyberattacks. Consequently, properly handling rekeying procedures for these *CHILD_SA*s is essential.

In the legacy *ipsec.conf* file, several options were available to manage how long the *SA* could remain active or how much data, in terms of *bytes* or *packets*, could be processed before a rekey or expiration event occurred. Even if these functionalities have been carried over to the new *swanctl.conf* format, some refinements have been implemented, meaning there is not always a direct mapping between the legacy and new options. In the *ipsec.conf* file, these parameters were organized as follows:

- *Time*: managed by the `lifetime` and `margintime` options. `lifetime` defined the maximum duration an *SA* could remain active before expiration, while `margintime` specified how far in advance the system should start renegotiating the *SA* parameters.
- *Bytes*: managed by the `lifebytes` and `marginbytes` options. `lifebytes` specified the maximum number of bytes transmitted over an *SA* before expiration, and `marginbytes` defined how far in advance the system should begin renegotiation.
- *Packets*: managed by the `lifepackets` and `marginpackets` options. `lifepackets` defined the maximum number of packets transmitted over an *SA* before expiration, and `marginpackets` specified how far in advance the renegotiation should begin.

Another option adopted to reduce predictability in the rekeying process is the `rekeyfuzz` parameter. This parameter is a percentage value applied to the `margin*` options to introduce variability, ensuring that renegotiation timings are not constant and less predictable. In the *swanctl.conf* file, these options have been updated:

- *Time*: managed by the **rekey_time**, **life_time**, and **rand_time** options. `rekey_time` defines the time threshold for renegotiation, and `life_time` functions like the original `lifetime`. `rand_time` introduces a random component to subtract from `rekey_time` to prevent simultaneous rekeys on both sides. To mirror the *ipsec.conf* behavior, these parameters should be configured as `rekey_time=lifetime-margintime`, `life_time=lifetime`, and `rand_time=margintime*rekeyfuzz`.
- *Bytes*: managed by the **rekey_bytes**, **life_bytes**, and **rand_bytes** options. `rekey_bytes` specifies the byte threshold for renegotiation, and `life_bytes` behaves like the original `lifebytes`. `rand_bytes` introduces variability by subtracting a random number of bytes from `rekey_bytes`. To replicate the *ipsec.conf* behavior, these parameters should be configured as `rekey_bytes=lifebytes-marginbytes`, `life_bytes=lifebytes`, and `rand_bytes=marginbytes*rekeyfuzz`.
- *Packets*: managed by the **rekey_packets**, **life_packets**, and **rand_packets** options. `rekey_packets` determines the packet threshold for renegotiation, while `life_packets` matches the functionality of `lifepackets`. `rand_packets` introduces a random variation by subtracting a random number of packets from `rekey_packets`. To achieve similar behavior to *ipsec.conf*, these parameters should be configured as `rekey_packets=lifepackets-marginpackets`, `life_packets=lifepackets`, and `rand_packets=marginpackets*rekeyfuzz`.

### 5.1.2 Firewall Management

When configuring an *IPsec VPN*, it is essential to ensure that firewall rules are set up correctly to allow *VPN* traffic to pass through the system's security policies, as without proper firewall configuration, *VPN* traffic may be blocked, rendering the *VPN* connection unusable. *strongSwan* has provided proper options to handle such scenarios.

In the legacy *ipsec.conf* structure, the `leftfirewall` option was used to ensure that basic firewall rules were automatically applied when a tunnel was established; this carried a boolean value to invoke an `_updown` default script to set up basic firewall rules.

In the context of *swanctl.conf*, the approach has changed. The **updown** option is the one responsible for the script invocation and is related to a specific plugin (*updown*) which allows the invocation of a script when an *IKEv2 CHILD_SA* gets established or deleted. More specifically, it is used to specify the path of the script to be executed under these circumstances, with eventual parameters immediately following and space-separated.

By default, there is an *_updown* script located at `/usr/libexec/ipsec/_updown`, which is designed to set up firewall rules using *iptables* to ensure *VPN* traffic is allowed through the system's security policies. To invoke this script, the `updown` option to the absolute path of the default *_updown* script should be set, also passing the `iptables` argument to ensure the script uses *Netfilter* to create rules; for instance, this might get achieved by setting `updown = /usr/libexec/ipsec/_updown iptables`. Note that if a configuration requires more complex or custom behavior, it is possible to realize a customized script and invoke it employing the same options; moreover, the phase of *CHILD_SA* rekeying is not considered as an establishment or removal of an association, meaning that the script will not be invoked under this circumstance.

### 5.1.3 Policies Management

In the context of *IPsec*, policies determine how traffic should be processed, specifically which traffic should be encrypted, decrypted, or bypassed. As anticipated in earlier sections, *IPsec* policies are typically managed by the *IKE* daemon installed in the kernel's *SPD (Security Policy Database)*.

In the *swanctl.conf* file, it is possible to define an option related to policy management, which is called **policies** (this has been carried from the legacy ipsec.conf file, but the option name was `installpolicy`); it is a boolean option used to control whether *IPsec* policies have to be installed for a child connection; disabling it by setting `policies = false` can be useful in scenarios where the policies are not supposed to be managed by the *IKE* daemon but rather by an external entity. For instance, for *Mobile IPv6*, policy decisions are handled separately, and there might be a need to avoid double handling or conflicts with the *IKE* daemon decisions.

### 5.1.4 Traffic Marking Management

Traffic marking is a technique for tagging or classifying network packets for special handling. Once packets are marked, the system can apply specific rules or policies based on these marks.

*strongSwan* makes available a series of options to apply specific marks on incoming or outgoing traffic; these are used to apply some form of control over traffic and can be set in the *swanctl.conf* file. The options related to inbound traffic are **mark_in**, **mark_in_sa** and **set_mark_in**, while the ones related to outbound traffic are **mark_out** and **set_mark_out**.

It is to be specified that these options are related to *Netfilter*, which is a Linux framework for packet filtering, *NAT (Network Address Translation)* and other packet manipulation tasks; it is used for many firewall solutions on Linux. The details of each are described here:

- `mark_in`: it is used to apply a *Netfilter* mark and mask for input traffic before it gets processed by the inbound *IPsec SA*; this is done to differentiate and manage incoming packets based on predefined rules to facilitate routing or policy application before decryption.

- `mark_in_sa`: used to determine whether the mark_in should also be applied directly to the inbound *SA*; this ensures that the mark_in persists post-decryption, allowing for continued handling based on the mark after the packets have been decrypted (by default, this is not applied).

- `set_mark_in`: used to apply a Netfilter mark to packets after the inbound *IPsec SA* has processed them; it is used when necessary to enable specific post-processing policies.

- `mark_out`: used to apply a Netfilter mark to outgoing traffic before the outbound *IPsec SA* encrypts it; it can be used to manage outgoing packets according to predefined rules before encryption.

- `set_mark_out`: used to apply a Netfilter mark to packets after the outbound *IPsec SA* has encrypted them; used for handling of encrypted packets according to post-encryption conditions.

In the legacy *ipsec.conf*, traffic marking was handled differently, with only essential support for marking traffic before and after encryption using the mark option. There was no distinction between pre-encryption and post-decryption marking, and no specific options like `mark_in_sa` or `set_mark_in/set_mark_out` existed.

### 5.1.5 Grouping problem

While the general transition from legacy capabilities to newer ones for channel protection has mainly been achievable using the existing structure and logic in the framework, a significant issue emerged during the analysis of approaches related not only to more recent channel protection implementations but also to other branches of cybersecurity, as documented in *Marchitelli*'s work [11].

The core issue involves managing capabilities that always appear in *groups*, meaning that certain capabilities are logically linked and must be processed together. To clarify the situation, consider the following example:

```
ike {
    ...
    local_addrs=192.168.0.100
    local {
        auth=eap-tls
        certs=CarolCert.pem
    }
    version=2
    ...
}
```

In this snippet from a `swanctl.conf` file for *strongSwan*, the `local_addrs` and `version` options are related to the general `ike` connection, whereas the `auth` and `certs` options appear within the scope of a specific subsection. As explained in the *Background*, this distinction is crucial for defining the logical structure of the configuration.

In the legacy approach of the *translator*, capabilities were processed as they appeared in the abstract rule artifact. This approach was sufficient for previous cases where similar issues had arisen in other *NSFs*, as the impact was minor and could be handled with slight adjustments. However, this approach proves inadequate in the current scenario, as the new requirements necessitate a more sophisticated solution that upholds the thesis's guiding principle of maintaining abstraction without involving low-level details.

Consequently, forcing a user to include capabilities in a specific order within the *XML Rule* artifact to align with the configuration specifics of *strongSwan* would violate the core principle of abstraction. The challenge, therefore, was to define a translation process that allows the abstract rule to remain order-agnostic while implementing a refinement procedure that respects the logical grouping of related capabilities.

The following sections analyze the steps taken and the structures adopted to address this problem, offering detailed insights into the technical aspects and implementation strategies used.

## 5.2 Groups Management

As mentioned in the previous section, one challenge was managing sets of capabilities that share a common logical purpose. For example, in the context of *strongSwan*, some capabilities are used to specify authentication details for the local peer, while others, with the same technical name, serve the same purpose but are applied to remote peers. To illustrate this issue, the following practical example is presented:

```
ike {
    ...
    local {
        auth=eap-tls
        certs=CarolCert.pem
    }
    remote {
        auth=eap-tls
        id="C=CH, O=strongSwan Project, CN=moon.strongswan.org"
    }
    ...
}
```

In this *strongSwan* configuration snippet, it is evident that some options are associated with the `local` group, while others pertain to the `remote` group. Some of these options even share the same key name (such as *auth* in this case). Initially, the framework did not provide a method to process these capabilities in such cases because it lacked a mechanism to inform the *translator* in advance that specific capabilities belong to distinct groups, which should be handled differently.

While it is straightforward at a low level to construct the *strongSwan* configuration by appropriately setting these options (as discussed in the *Background* section), it would be inappropriate to expect users of the *Security Capability Manager* framework to consider such divisions when writing high-level configuration files. This approach would compromise the abstraction principle, as implementation-specific details would become exposed in high-level files. Furthermore, enforcing a static and universally valid grouping of capabilities within the framework is impractical, as different *NSFs* may implement the same capabilities but follow different grouping logics.

A viable solution to this challenge lies in the `NSFPolicyDetails` element. As previously discussed, these details provide specific information about how a generic rule for a single *NSF* should be translated into the framework's catalogue. It is, therefore, appropriate to introduce the necessary modifications into this component to handle groups, allowing the framework to manage grouping peculiarities of each *NSF* while maintaining an agnostic approach for the user. This ensures that translations occur automatically, adapting to the grouping logic of the target *NSF*.

### 5.2.1 Model and Catalogue update

The first step in introducing support for groups was updating the *Capability Information Model* at the level of the `NSFPolicyDetails` class. Specifically, a new class was introduced to describe the possible groupings of capabilities for each *NSF*:

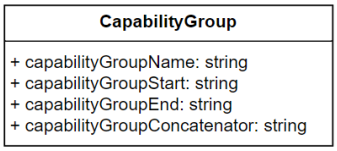


**Figure 5.1:** Capability Group

This update allows the definition of a capability group with a `name` used in the translation process, a `start` and `end` string to delimit the group, and a `concatenator` string to specify how the capabilities within the group should be separated. In terms of relationships, the `GroupCapability` class is linked to the `SecurityCapability` class through an *aggregation* relationship, where `GroupCapability` serves as the container class, and the group is defined as an aggregation of one or more security capabilities. Similarly, `GroupCapability` is also linked to the `NSFPolicyDetails` class through another *aggregation* relationship, with `NSFPolicyDetails` acting as the container class, allowing the policy details for an *NSF* to be defined as an aggregation of one or more groups of capabilities.

Regarding the `NSFCatalogue`, one or more `capabilityGroup` tags can now be defined within the `nsfPolicyDetails` element, following the model's validation, containing details about the groups. Below is an example of a group defined for *strongSwan*:

```
<nsfPolicyDetails>
    ...
    <capabilityGroup>
        <capabilityGroupName>local</capabilityGroupName>
        <capabilityGroupStart>local {\n\t\t</capabilityGroupStart>
        <capabilityGroupEnd>}\n\t</capabilityGroupEnd>
        <capabilityGroupConcatenator>\t</capabilityGroupConcatenator>
        <securityCapability ref="SourceAuthActionCapability"/>
        <securityCapability ref="SourceIdentityConditionCapability"/>
        ...
    </capabilityGroup>
    ...
</nsfPolicyDetails>
```

As shown, aside from the `name`, `start`, `end`, and `concatenator` strings mentioned above, the security capabilities that will be part of the group must be listed sequentially, drawing from the complete list of capabilities in the corresponding *NSF* element of the catalogue.

### 5.2.2 Including the Dependency Tree

Once the definition of the groups is provided in the catalogue, the following step was to explore ways to include them in the translation process so that it could be enhanced and support the new cases of grouped capabilities. To reach the goal, this thesis leaned on the parallel work by *Marchitelli*, who implemented a structure called **Dependency Tree** at the level of the details generation for each *NSF* [11].

The *dependency tree* has been added in the framework to solve a series of issues in the legacy system regarding dependencies between different capabilities; in brief, some already supported *NSF*s exhibit capabilities that, for a syntactically correct rule generation, depend on other capabilities' presence or values to deduce how they must be translated. With this structure, *Marchitelli* introduced a dynamic approach to translating the rule before the actual translation process partially [11].

This is done by generating a tree that starts with a particular list of all the possible *Command Names* of the *NSF*'s capabilities; then, at each node, the dependencies of the involved capabilities are processed to deduce how that list would change if they were respected or not in an eventual real rule. The generation ends with leaf nodes containing a list of *Command Names* for the *NSF*'s capabilities concerning their dependencies; the final result is a binary tree that contains, at the level of the leaves, all the possible *POSET*s for the *NSF*, according to each capabilities' dependencies and if these are respected or not. For a more specific view on how each *POSET* is generated and how the dependencies are checked and used in the scope of the *dependency tree*, the relative work from *Marchitelli* can be consulted [11].

For this thesis, the *dependency tree* has proven crucial as the *POSET*s represent a temporary but faithful version of the rule that must come out at the end of the translation process; consequently, letting the groups' details into the generation flow of the tree appeared as the best solution, so to make the *POSET*s in the leaf nodes already structured according to the groups' structures of the related *NSF*.

### 5.2.3 Generation of Groups' Details

The most appropriate point to insert the reading of group details from the catalogue was determined to be at the level of the *root node* of the *dependency tree*. Given the nature of its *commandName* elements, a straightforward approach was to maintain the structure of the existing class. Practically, the delimiters of the groups are visualized within the *dependency tree* as *Command Names*, similar to those associated with traditional capabilities but with unique properties. This ensures that when *POSETs* for the *NSF*s are generated, the group delimiters will be automatically injected, constraining the capabilities within the corresponding group.

A new class, `GroupDetails`, has been introduced to store the relevant information. In addition to several string type attributes for saving group details (`capabilityGroupName`, `capabilityGroupStart`, `capabilityGroupEnd`, and `capabilityGroupConcatenator`), the class includes an attribute `capabilityGroupMembers` of type `List<String>` to store the list of capability names contained in the group.

Below is an explanation of the functions introduced to extract group details and save them as objects of the `GroupDetails` class for the tree generator (note that they are gathered in the `DependencyTreeManager.java` file):

- `createDetailsForEachNSFGroup`: this function is invoked by the `generateRootNode` function and iterates through the catalogue to retrieve the `NSFPolicyDetails` element for the requested *NSF*. Once retrieved, it iterates over all `capabilityGroup` elements, invoking the `addDetailsForSpecificNSFGroup` function to obtain the details for each specific group. It concludes by returning the list of details for all groups.

- `addDetailsForSpecificNSFGroup`: this function is invoked by `createDetailsForEachNSFGroup` and retrieves the delimiters for each group, as well as the list of capabilities contained in it. It concludes by returning a newly created `GroupDetails` object, which will be inserted in a variable of type `List<String>` by the calling function.

### 5.2.4 Including the Capability Processing Order

With the groups' details gathered into appropriate structures, the following requirement was to inject them into the primary workflow of the framework. The first challenge was understanding where and how to manipulate the flow to consider the groups' details and have them inserted into the *POSET*s of the *dependency tree*, as stated before.

After some trials, the best solution appeared to lie in another structure, called **Capability Processing Order**, introduced by *Marchitelli*, whose work can again be consulted for more extensive explanations [11]. For this thesis, it should be considered that this variable's purpose is to contain a preliminary order with which the *NSF*'s capabilities must be processed in the translation procedure. Specifically, it is intended as a string representing a serialized list of lists, each specifying a subset of the *NSF*'s capabilities in an order that must be respected when printed in the final rule; the capabilities in a single list that must be printed in a particular order are comma-separated, while the different lists, independent from each other, are semicolon-separated.

For instance, if capabilities `cap1`, `cap2` and `cap3` must be printed in this sequence, while `cap4` and `cap5` must also be printed in sequence, but independently from the first three, the `CapabilityProcessingOrder` variable will be `cap1,cap2,cap3;cap4,cap5`.

### 5.2.5 Merging Group Details and Capability Processing Order

Leveraging the `CapabilityProcessingOrder` element and its design, the natural progression of the group implementation was, as anticipated, to integrate group details into this structure. This allows the capabilities of specific groups to be gathered when the string containing the processing order is expanded with all capabilities and the group delimiters to be seamlessly inserted where appropriate.

The general approach for this procedure involves taking the `CapabilityProcessingOrder` from the catalogue, if available, and expanding the sub-lists of the capabilities so that all eventual group delimiters are included. It is important to note that, due to the structure of the *NSFs* supported by the framework, lists of capabilities requiring a specific order that belongs to different groups are not supported (an *exception* is thrown if this situation is detected). Finally, all the groups without a predefined processing order are added to the end of the processing order, along with capabilities that do not belong to any group. The result is a `CapabilityProcessingOrder` that contains all the capabilities of the *NSF*, grouped according to delimiters while maintaining the original processing order where specified.

For example, it can be supposed that the framework's catalogue supports an *NSF* which has the following details regarding groups:

- A group `Group1`, with *start*, *end* and *concatenator* strings equal to `start1`, `end1` and `con1`, and containing capabilities `cap1` and `cap2`.
- A group `Group2`, with *start*, *end* and *concatenator* strings equal to `start2`, `end2` and `con2`, and containing capabilities `cap3`, `cap4`, and `cap5`.
- Capabilities `cap6` and `cap7` that are not part of any group.

If the `CapabilityProcessingOrder` is `cap2,cap1;cap3,cap5`, the final order is `start1,cap2,con1,cap1,end1;start2,cap3,con2,cap5,con2,cap4,end2;cap6;cap7`. This ensures that the partial group orderings are expanded and the missing capabilities are appended at the end without imposing a specific order; furthermore, the processing order has been expanded with all the delimiters for the specific groups.

Below is an overview of the new functions introduced to process the capability processing order, with group management taken into account:

- `expandCapabilityPrintOrder`: this function is a wrapper for all the subsequent functions mentioned. It is invoked by `generateRootNode` and expands the `capabilityPrintOrder` element to include all possible capabilities of the *NSF* while maintaining the partial order provided initially. It concludes by returning the capability processing order as a string containing all the capabilities appended in the correct order.

- `initializeAllListOfCapabilitiesWithOrder`: this function receives a series of capabilities, provided as strings, that must maintain a particular order according to the initial `capabilityProcessingOrder`. It transforms them into lists for easier processing. It is invoked by `expandCapabilityPrintOrder`, and the individual lists are assumed to contain capabilities from the same group, which will be assessed in a later function.

- `addAbsentGroupMembers`: this function takes the lists of capabilities with predefined ordering and expands them by appending the remaining capabilities of the same group. It is invoked by `expandCapabilityPrintOrder` and returns the expanded lists.

- `addRemainingGroupMembers`: invoked by `addAbsentGroupMembers`, this function concatenates the remaining group capabilities of a specific group. It also verifies that the capabilities in each partial ordering belong to the same group or no group at all, throwing an *exception* if an unsupported configuration is encountered in the processing order.

- `addRemainingGroups`: this function adds the capabilities of all the groups absent in the original `capabilityProcessingOrder`. It is invoked by `expandCapabilityPrintOrder`, and by its execution, the list contains complete capabilities for all groups of the specific *NSF*.

- `expandGroupWithConcatenators`: this function takes the lists of group capabilities produced by the previous functions and expands them by adding the group delimiters. It is invoked by `expandCapabilityPrintOrder`, and upon completion, all lists of group capabilities are finalized, including the respective delimiters.

- `addCapabilitiesNotInAnyGroup`: this function appends to the end of the capability processing order element any capabilities that were not part of the original processing order and do not belong to any group. It is invoked by `expandCapabilityPrintOrder`, and by the end of its execution, it returns the final processing order as a string containing all capabilities of the *NSF*.

Once this procedure is completed, the `expandCapabilityPrintOrder` function merges the lists produced into a single `finalCapabilityPrintOrder` string, returning to the initial format; this will be utilized by the processing logic of the *dependency tree* to define the *POSET*s, as already described. As a result, each *POSET* in the leaf nodes of the tree, for the requested *NSF*, will be constructed taking into account also the group structure and will include the command names for all the delimiters, making it ready for direct use in the translation process.

## 5.3 POSET Processing for Rule Translation

Once the *POSET*s for the target *NSF* have been generated and the group details injected into them, the next step is to consider the actual rule provided to the translation framework. Previously, the *translator* tool iterated over the rule's capabilities based on its high-level description. Now, the process has been modified. Since the *POSET* now contains the capabilities' command names as they should appear after translation, along with the delimiters for any groups, it is appropriate for the *POSET* structure to be used as the basis for iteration during the translation process instead of the abstract rule.

However, before this translation can occur, the *POSET* must undergo a filtering procedure. In its initial form, the *POSET* contains the capabilities and group structure of the entire *NSF*, based on respected dependencies, so removing all capabilities not present in the specific rule is necessary.

With these considerations in mind, the main workflow is now updated concerning the legacy system and is as follows:

1. Extract the correct *POSET* for the rule based on the *NSF* capabilities' dependencies and characteristics.

2. Filter the *POSET* to contain only the command names of the capabilities appearing in the abstract rule.

By the end of these steps, the abstract rule is ready to be translated into a low-level one, as in the previous framework version. However, thanks to the *POSET* structure, the translation no longer relies on the high-level rule's input file structure and incorporates grouping information where applicable.

### 5.3.1 Retrieve the correct *POSET*

In his parallel work, the first step of the filtering procedure of the *POSET* has been achieved by *Marchitelli* [11]. Specifically, in the `TranslatorUtils.java` file, the `translateRule` function, which is called in the main flow of the translation process, invokes the `getMatchingPoset` function; this function takes as inputs the capabilities of the rule to be translated and the tree node representing the *dependency tree*, and retrieves the appropriate *POSET* for that rule, according to a sequence of steps that involve checks on the capabilities dependencies.

Again, *Marchitelli*'s work can be consulted for details about how this retrieval process is carried out [11].

### 5.3.2 Filtering the *POSET*

As mentioned earlier, the correct *POSET* must now be processed to retain only the capabilities required to translate the rule. Generally, the approach involves iterating over the capabilities in the *POSET* and checking if each is present in the rule; if not, it is removed. The management of the group segments, however, requires additional processing. Once the absent capabilities are removed, the *concatenator* strings within the group sections must be adjusted by removing any redundant concatenators. Additionally, if a group section ends up with no capabilities, the *start* and *end* strings must also be removed from the *POSET*.

For example, consider a situation similar to the one described in the `capabilityPrintOrder` example. Assume the framework's catalogue supports an *NSF* with the following details:

- Group `Group1`, with *start*, *end*, and *concatenator* strings `s1`, `e1`, and `c1`, containing capabilities `cap1` and `cap2`, with the command names `cm1` and `cm2`, respectively.
- Group `Group2`, with *start*, *end*, and *concatenator* strings `s2`, `e2`, and `c2`, containing capabilities `cap3`, `cap4`, and `cap5`, with the command names `cm3`, `cm4`, and `cm5`, respectively.
- Capabilities `cap6` and `cap7`, not part of any group, with the command names `cm6` and `cm7`, respectively.

After processing, starting from a `capabilityProcessingOrder` element, assume that the *POSET* generated for a rule to be translated is `[[[s1], [cm2], [c1], [cm1], [e1]], [[s2], [cm3], [c2], [cm5], [c2], [cm4], [e2]], [[cm6]], [[cm7]]]`. If a rule provided to the framework includes the capabilities `cap3`, `cap4`, and `cap6`, the resulting filtered *POSET*, ready for translation, will be `[[[s2], [cm3], [c2], [cm4], [e2]], [[cm6]]]`.

Below is an overview of the functions implemented in the *translator* that perform this process (all of these functions are located in the `TranslatorUtils.java` file):

- `cleanPosetAccordingToRule`: This function is a wrapper for all the subsequent functions. It is invoked by the `translateRule` function and cleans a *POSET* based on a given rule and a list of capabilities. It extracts the command names from the *POSET*, then iterates through the capabilities, removing those not present in the rule. Group sections, identified if they start with a `groupStarter` element, are handled separately. Invalid or redundant elements are removed during the process, and the *POSET* is updated to include only the necessary capabilities. Finally, the function removes all empty groups and returns the filtered *POSET* as a set.

- `handleGroupSegmentOfPoset`: This function processes a specific section of the *POSET* that represents a group of capabilities starting with a `groupStarter` element. It is called by the `cleanPosetAccordingToRule` function and collects all elements belonging to the group. It then processes the group to identify valid capabilities by comparing them with the rule's allowed capabilities. Any capability not in the valid list is flagged for removal. After removing invalid entries, the function cleans up redundant `groupConcatenator` elements and returns the cleaned group. If no valid capabilities remain in the group, the group is discarded.

- `addInGroupSectionAllPossibleCapabilities`: This function is invoked by `handleGroupSegmentOfPoset` and iterates through the remaining capabilities in the *POSET*, adding them to a variable that stores all the capabilities of the group that appear sequentially in the *POSET* until it encounters a `groupEnder` capability. It ensures that all capabilities belonging to a group, as marked by group boundaries, are included in the group section for further processing. This function effectively delimits the group of capabilities within the *POSET*.

- `identifyCapabilitiesInRule`: This function is invoked by `handleGroupSegmentOfPoset` and identifies and counts valid capabilities within a group based on a predefined list extracted from the rule. It checks each capability in the group against the list of valid capabilities. If a capability is not present in the rule or is a group delimiter, it is marked for removal. The function returns the number of valid capabilities found, which is used later to decide whether the group should be retained or discarded.

- `removeRedundantConcatenatorsInGroupSection`: This function, invoked by `handleGroupSegmentOfPoset`, cleans up redundant `groupConcatenator` elements within a group section in the *POSET*. Since capabilities have been removed without considering previous and following concatenators, this function eliminates any consecutive concatenators left after capability removal, leaving only the necessary ones.

- `cleanPosetOfEmptyLists`: This function, invoked by `cleanPosetAccordingToRule`, iterates through the *POSET* and removes any empty lists of capabilities. Its primary purpose is to finalize the filtering of the *POSET* after processing, ensuring no empty groups remain.

At the end of this process, the *POSET* is ready for the translation procedure using an approach that will be presented in the following section.

## 5.4 POSET Translation

With the *POSET* prepared and containing only the command names of the capabilities from the abstract rule, the final step is to enhance the logic previously implemented at the *translator* level. Specifically, two main changes were required:

- *Iteration over POSET*: In the earlier version of the *translator*, capabilities were iterated based on their order in the abstract rule. It is necessary to iterate over the capabilities as they appear in the *POSET*. This iteration ensures that the order of processing aligns with the structure defined by the *POSET*, reflecting dependencies and capabilities groupings are considered.
- *Retrieval of the string details for the capability*: Previously, all low-level strings for each capability were directly retrieved from the *NSF catalogue*. In the updated version, these strings are now retrieved from the `commandName` structures in the *POSET*.

The first change has been implemented in the `Translator.java` file, specifically within the `translate` function. After completing the preliminary setup, the original iteration over the `nodes` collection (which represents the capabilities in the *XML Rule* artifact) has been updated to iterate over the `CommandName` elements in the *POSET*. Consequently, the capabilities are processed according to the sequence defined in the *POSET*. One critical adjustment is that preliminary checks must be conducted to identify these elements since the *POSET* also contains dummy `CommandName` elements for the group delimiters. These elements are processed differently by directly retrieving the delimiter value and inserting it into the partially translated rule.

The second change has been implemented in the `ClausePreTranslator.java` file, specifically in the `getSupportRuleAttributeString` function. In the legacy version of the *translator*, the low-level strings for each capability (such as its specific name for the target *NSF*) were retrieved by accessing the *NSF catalogue*. In the new implementation, the `commandName` elements introduced by *Marchitelli* in the framework are now directly used, as they are also the elements present in the *POSET* [11]. These `commandName` elements carry all the necessary details for translation, making it sufficient to query each capability's `commandName` for the required information.

Once these final adjustments have been completed, the implementation work has ended to enhance the translation process with all the new functionalities presented in previous sections. Specifically, the *POSET* structure has been injected into the process and used for the translation procedure while integrating dependencies and groupings for the various capabilities. The following chapter will provide validation examples for these changes through *strongSwan*.

# Chapter 6

# Solution Validation

This chapter outlines the validation procedure used during the implementation phase. As previously mentioned, *strongSwan* has been adopted as the primary tool, and several configurations from the official documentation have been utilized as validation benchmarks. Validation operations were conducted multiple times throughout the thesis work in alignment with modern development methodologies. This iterative approach ensured that the framework's behavior was assessed at the end of each milestone that introduced new features or changes. The following are the stages at which validation was performed:

1. *Design Phase*: This stage involved examining general configurations for channel protection and mapping them onto the *UML* model constructed during this phase.
2. *Base Capabilities Support*: Validation was conducted after enhancing the legacy capabilities already present in the framework for channel protection.
3. *Insertion of Group Details*: For this step, it was essential to verify that definitions of groups for different *NSFs* were correctly established and that their subsequent retrieval from the *NSF Catalogue* functioned as intended.
4. *Manipulation of the POSET*: At this stage, validation focused on enhancing the `capabilityProcessingOrder` variable and ensuring its propagation into the *POSET*, including the incorporation of group details.
5. *Filtering of the POSET*: During this step, validation ensured that irrelevant capabilities were appropriately removed and that the remaining elements conformed to the intended structure of the rule.
6. *Translation Through the POSET*: The final stage of the implementation phase required validating the translated rule against real-world configurations to confirm its accuracy.

Each of these points will now be discussed in more detail.

## 6.1 Design Phase Validation

After defining the *UML* model for channel protection, an initial validation process was conducted, even if performed at a high level. This validation aimed to determine whether the model could effectively represent all significant aspects of actual security configurations while abstracting away low-level implementation details, which aligns with this thesis's objectives. To achieve this, *strongSwan* configurations were used, similar to the approach adopted for the final validation. A simple notation was employed to verify that the security properties of each configuration could be accurately mapped onto the *UML* model. Specifically, this notation utilized an *OOB*-like dotted format to access and reference different classes within the model.

Below is an example of the configurations used for this validation step. In particular, it represents a straightforward *strongSwan* scenario involving two hosts establishing a secure communication channel in *tunnel* mode. The setup utilizes *public-key certificates* for mutual authentication, along with selecting appropriate algorithms for encryption, integrity, and other security operations, with additional settings to customize the connection.

```
connections {
   host-host1 {
      local_addrs  = 192.168.0.1
      remote_addrs = 192.168.0.2
      local {
         auth = pubkey
         certs = moonCert.pem
         id = moon.strongswan.org
      }
      remote {
         auth = pubkey
         id = sun.strongswan.org
      }
      children {
         host-host2 {
            updown = /usr/local/libexec/ipsec/_updown iptables
            rekey_time = 5400
            rekey_bytes = 500000000
            rekey_packets = 1000000
            esp_proposals = aes128gcm128-x25519
            mode = transport
         }
      }
      version = 2
      mobike = no
      reauth_time = 10800
      proposals = aes128-sha256-x25519
   }
}
```

This configuration is then mapped to the *UML* model using the previously mentioned notation to illustrate how each security property aligns with the model's structure.

```
ChannelProtection: host-host2
    .AETechniqueCapability:
        .Algorithm: aes128
        .Mode: gcm
        .IntegrityCheckLength: 128
    .TunnelEncapsulationCapability
        .PeerLocal: 192.168.0.1
        .PeerRemote: 192.168.0.2
    .rekey_time: 5400
    .rekey_bytes: 500000000
    .rekey_packets: 1000000
    .updown = /usr/local/libexec/ipsec/_updown iptables
    .KeyMaterialCapability
        .DynamicKeyExchangeCapability
            .EllipticCurveDHExchangeCapability: x25519
            .ChannelProtection: host-host1
                .EncryptionTechniqueCapability:
                    .Algorithm: aes128
                    .Mode: cbc
                .PayloadAuthCapability
                    .Hash-BasedAuthMethodCapability
                    .HashFunction: sha256
                    .Mode: hmac
                .KeyMaterialCapability
                    .EllipticCurveDHExchangeCapability: x25519
                    .PeerLocal
                        .RSAAuthenticationCapability
                            .DNSName: moon.strongswan.org
                            .certs: moonCert.pem
                    .PeerRemote
                        .RSAAuthenticationCapability
                            .DNSName: sun.strongswan.org
                    .reauth_time: 10800
                    .version: 2
                    .mobike: no
```

Note that some specific capabilities, namely the `rekey_*` options, `updown`, `reauth_time`, `version`, and `mobike`, do not have a direct equivalent in the model described in the chapter on the design phase. This omission is intentional, as the primary goal was to model only the most essential aspects of channel protection, as detailed earlier. Still, these capabilities were incorporated during the implementation phase when their addition required a refactoring or expansion of capabilities that the model already supported.

After numerous examples like the one previously presented were successfully mapped, it was confirmed that the model effectively captured the essence of channel protection across a wide range of scenarios.

## 6.2 Base Capabilities Support Validation

Providing support for new or enhanced capabilities has been achieved by updating the *NSF Catalogue* accordingly. These initial expansions were incorporated using the existing functionalities provided by the framework, and validation was carried out by executing the translation process as-is, ensuring that the generated options conformed to *strongSwan*'s syntax and conventions.

Below is an example of a new capability added to the catalogue to support rekeying procedures:

```
<capabilityTranslationDetails>
    <nSF ref="StrongSwan"/> <securityCapability ref="
    LifePacketsConditionCapability"/>
    <commandName
        <realCommandName>rekey_packets</realCommandName>
    </commandName>
    <internalClauseConcatenator>=</internalClauseConcatenator>
    <clauseConcatenator>\n\t</clauseConcatenator>
</capabilityTranslationDetails>
```

To validate this capability, the *XML Rule* artifact was used by adding the following high-level requirement:

```
<lifePacketsConditionCapability>
    <value>1000000</value>
</lifePacketsConditionCapability>
```

After the translation process, the resulting output file contained the following option, which adheres to *strongSwan*'s specific syntax:

```
rekey_packets=1000000
```

This output demonstrates that the framework successfully mapped the high-level specification to the correct low-level configuration following *strongSwan*'s requirements. All other changes made to the *NSF Catalogue* were validated similarly, confirming that each capability was correctly implemented and mapped, consequently supporting an *NSF* focused on channel protection. At this time of the implementation, capabilities were tested only singularly, without merging them into a complete and valid rule (this was due to the still missing support for groups).

## 6.3 Insertion of Group Details Validation

Reading and saving the groups' details into appropriate structures required validation, particularly at the level of the `groupDetailsList` variable. This variable is a `List<GroupDetails>` structure designed to store the characteristics of each group after the procedures described in the *Implementation Phase* have been completed. Direct testing was conducted for *strongSwan* first to ensure that the *NSF Catalogue* could be appropriately updated to include the group details. Subsequently, verifying that this information was correctly retrieved and represented within the framework was necessary.

The primary validation approach involved including *strongSwan*'s groups. This was followed by testing whether the updated details could be successfully processed. Below is a snippet of the `NSFPolicyDetails` for *strongSwan*, updated to include the new information:

```
<nsfPolicyDetails>
    ...
    <capabilityGroup>
        <capabilityGroupName>local</capabilityGroupName>
        <capabilityGroupStart>local {\n\t\t</capabilityGroupStart>
        <capabilityGroupEnd>}\n\t</capabilityGroupEnd>
        <capabilityGroupConcatenator>\t</capabilityGroupConcatenator>
        <securityCapability ref="SourceAuthActionCapability"/>
        <securityCapability ref="SourceIdentityConditionCapability"/>
        <securityCapability ref="AaaIdentityConditionCapablity"/>
        <securityCapability ref="XAuth_identityConditionCapability"/>
        <securityCapability ref="SourceEapIdentityConditionCapability"/>
        <securityCapability ref="
    SourceX509CertificateConditionCapability"/>
    </capabilityGroup>
    ...
</nsfPolicyDetails>
```

The result obtained during the workflow and stored into `groupDetailsList` is:

```
[{capabilityGroupName='local', capabilityGroupStart='local {\n\t\t',
    capabilityGroupEnd='}\n\t', capabilityGroupConcatenator='\t',
    capabilityGroupMembers [SourceAuthActionCapability,
    SourceIdentityConditionCapability, AaaIdentityConditionCapablity,
    XAuth_identityConditionCapability,
    SourceEapIdentityConditionCapability,
    SourceX509CertificateConditionCapability]}, ...]
```

It can be seen that the delimiters and the members of the *local* group have been correctly retrieved, alongside all the other groups omitted here for space convenience.

## 6.4 Manipulation of the POSET Validation

With the groups' details adequately stored, the next major validation step occurred after the *POSET* was manipulated through the `capabilityProcessingOrder`. Custom tests, including dummy `capabilityProcessingOrder` variables, were executed to evaluate potential corner cases and ensure robustness in handling different scenarios. In the case of *strongSwan*, no particular processing order is required, which allows for testing some aspects of its pre-processing separately. Nevertheless, the correctness of the `finalCapabilityProcessingOrder` still needed to be verified.

After pre-processing the capability processing order, the `finalCapabilityProcessingOrder` for *strongSwan* was generated, containing a value representing all the groups with their respective capabilities and delimiters. Additionally, any capabilities not belonging to a group were placed at the end of the variable, consistent with the intended design and objectives outlined in the related*section*.

Building upon this result, the subsequent generation of the *dependency tree* successfully incorporated the updated processing order. As a result, all the *POSET*s reflected the correct processing sequence, confirming that they were ready for enhancing the translation process, as described in the *Implementation Phase*.

## 6.5 Filtering of the POSET Validation

After filtering according to the abstract rule, the subsequent validation phase focused on the manipulated *POSET*. This phase involved providing various *XML Rule* artifacts representing *strongSwan* configurations and verifying that the resulting *POSET* contained all and only the *Command Name* elements of the abstract capabilities present in those artifacts. This ensured the filtering process accurately reflected the intended high-level rule, with no extraneous or missing elements.

The specific point for performing this validation was in the `TranslatorUtils.java` file, where the `posetReadyForRule` variable was tracked. This variable, according to the outcomes of the tests, effectively stores the output of the `cleanPosetAccordingToRule` function, which removes irrelevant capabilities from the *POSET* and retains only those that appear in the provided abstract rule.

It is important to note that, in this context, the practical applicability of the provided abstract rule was still not considered. The focus was solely on the correctness of the *POSET* filtering process, ensuring that the abstract capabilities were represented in the resulting structure regardless of logical consistency between them; this will be addressed in the last validation phase.

## 6.6 Translation Through the POSET Validation

In this section, the final validations, considered the most significant in achieving this thesis's main goals, will be described. After modifying the *translator* tool, it was crucial to assess realistic scenario rules and verify that by providing abstract rules as inputs, the framework could correctly generate rigorous and functional low-level rules for *strongSwan*. These final validations ensured that the abstract configurations, when representing complete and usable configurations, could effectively be translated into precise implementations.

As mentioned at the beginning of this chapter, the configurations used for validation were derived from the official *strongSwan* documentation and repository. Particular emphasis was placed on scenarios involving different types of peer authentication, such as using *PSK*, *EAP*, or *public key certificates*, as well as on building connections for both *peer-to-peer* communication and *VPN* setups. Additionally, these tests incorporated customized details, including rekeying procedures, firewall instantiations, and various security algorithms, to evaluate the flexibility and accuracy of the framework.

For illustrative purposes, below is an example configuration from a scenario involving a gateway with a specific *IP* address on its external interface toward the network. This gateway performs authentication using a *public key certificate* and accepts connections from remote peers that authenticate using *public key certificates* as well. The *strongSwan* configuration reported is the final output of the translation process, which is identical to the original one except for the order of the options inside the groups. This is normal due to the nature of the *POSET* (a set is not inherently ordered) and is acceptable because only groups must be respected, while single capabilities inside them can be in any disposition.

```
ike-name-placeholder {
    remote {
        auth=pubkey
    }
    local_addrs=192.168.0.1
    proposals=aes128-sha256-x25519
    local {
        auth=pubkey
        id=moon.strongswan.org
        certs=moonCert.pem
    }
    children {
        child-name-placeholder {
            esp_proposals=aes128gcm128-x25519
            updown=/usr/local/libexec/ipsec/_updown iptables
            local_ts=10.1.0.0/16
        }
    }
    version=2
}
```

Here follows the abstract rule that was provided to the framework. With this validation step at its end, it is proven that low-level configurations can also be replicated for valid channel protection configurations through proper abstraction that never considers the *NSF*'s details.

```
<rule id="0">
    <ipSourceAddressConditionCapability operator="exactMatch">
        <capabilityIpValue>
            <exactMatch>192.168.0.1</exactMatch>
        </capabilityIpValue>
    </ipSourceAddressConditionCapability>
    <sourceAuthActionCapability>
        <method>pubkey</method>
    </sourceAuthActionCapability>
    <sourceX509CertificateConditionCapability>
        <path>moonCert.pem</path>
    </sourceX509CertificateConditionCapability>
    <sourceIdentityConditionCapability>
        <identity>moon.strongswan.org</identity>
    </sourceIdentityConditionCapability>
    <destinationAuthActionCapability>
        <method>pubkey</method>
    </destinationAuthActionCapability>
    <sourceSubnetConditionCapability operator="rangeCIDR">
        <capabilityIpValue>
            <rangeCIDR>
                <address>10.1.0.0</address>
                <maskCIDR>16</maskCIDR>
            </rangeCIDR>
        </capabilityIpValue>
    </sourceSubnetConditionCapability>
    <sourceFirewallActionCapability>
        <choice>/usr/local/libexec/ipsec/_updown</choice>
    </sourceFirewallActionCapability>
    <channelProtectionConditionCapability>
        <cipherSuite>
            <algoEnc>aes128</algoEnc>
            <mode>gcm128</mode>
            <dhgroup>x25519</dhgroup>
        </cipherSuite>
    </channelProtectionConditionCapability>
    <exchangeVersionActionCapability>
        <version>2</version>
    </exchangeVersionActionCapability>
    <keyMaterialConditionCapability>
        <encAlgo>aes128</encAlgo>
        <hashAlgo>sha256</hashAlgo>
        <dhChoice>x25519</dhChoice>
    </keyMaterialConditionCapability>
</rule>
```

# Chapter 7

# Conclusions and Future Works

This thesis has built its foundation on prior work by researchers and students who explored how abstracting security capabilities, provided by diverse *NSF*s, can embody a solution to emerging issues in today's cybersecurity landscape. The primary aim was to introduce a novel approach to security configurations that allows high-level descriptions of secure systems, which can be efficiently utilized by administrators managing these systems across different technologies.

By leveraging the principles of *NFV* and *SDN*, as well as *UML* modeling and best practices in software engineering, this work proposes an abstract model for channel protection. The model extracts the most significant aspects of this cybersecurity domain while ignoring the details specific to individual implementations. The overarching goal has been to relieve users from the burden of understanding and adhering to low-level syntax and conventions, which has repeatedly been shown to negatively impact the robustness and reliability of the information systems they aim to secure.

The actual implementation phase shifted towards an existing framework, the *Security Capability Manager*, which resulted from previous works and already supported *NSF*s related and non-related to channel protection for the same abstraction purpose. Some existing capabilities related to this domain have been updated, while new ones have been developed from scratch to support the most common procedures for securing communication channels under the framework. Although, within the scope of this thesis, translation details have been provided only for *strongSwan*, these capabilities are designed to be referenced by other *NSF*s, respecting the framework's guiding principles and ensuring broad and adaptable support for different implementations.

In response to the growing trend in many technologies to adopt scalable and flexible solutions, which is to organize functionalities in hierarchical and grouped structures, this work introduces a comprehensive implementation for grouping the capabilities of various *NSF*s into specific logical or functional domains.

The framework now supports defining multiple capability groups for a single *NSF* and indicating how to delimit these groups using appropriate markers. These details can be included in the *NSF Catalogue* and will be automatically processed and stored during the generation of the abstract language.

Building upon the work from *Marchitelli* [11], a *POSET* structure was integrated into the framework to manage dependencies between capabilities effectively. For this thesis, the *POSET* was further expanded to incorporate support for capability groups. As a result, the framework now leverages this enhanced structure to guide the translation process, effectively handling practical channel protection *NSFs*, specifically *strongSwan*.

The results obtained from this work have successfully met the requirements proposed for this thesis, establishing a formal model for channel protection capabilities and implementing the essential functionalities within the existing framework to ensure proper handling and translation of security configurations. The immediate benefit of this is for professionals working in increasingly complex cybersecurity environments: they can now adopt advanced methodologies and implement secure systems in an abstract and reusable manner, mitigating errors that arise solely from the complexities of numerous technologies and specific algorithmic implementations, which are becoming more and more numerous and diverse in today's technological landscape.

Building on these achievements, future work can further enhance the current framework by expanding the support available for *strongSwan* and extending it to other *NSF*s that focus on channel protection. As is, *Security Capability Manager* could receive further refinement in broadening its coverage for this security branch to support even more capabilities and manage more rarefied scenarios and configurations.

Finally, future research could start with how the formal model for secure communication channels has been implemented and obtain valuable insights into modeling other security systems. These systems could adhere to the same guiding philosophy demonstrated in this thesis and reach similar results for different security domains, which has proven effective through continuous validation phases conducted throughout the work.

# Appendix A

# Appendix A

This chapter gathers all the explanations about new functionalities from *strongSwan* that are less common and interesting in daily scenarios and, consequently, have not been implemented. Still, they have been documented for future work.

## A.1 Authentication Rounds

The **round** option in the *local* and *remote* connection sections of the *swanctl.conf* file are integer type options used to define the order in which authentication rounds are processed; the sorting of the authentication rounds can be then set explicitly if multiple authentication methods have been determined. If the round option is not specified by default, a value of `0` is assigned, and the system will process the authentication rounds based on their position in the configuration file.

Note that in the legacy *ipsec.conf* file, there was no direct correspondence of this option; it was only possible to define an authentication method used locally or required from the remote employing the `leftauth` and `rightauth` options, providing an alternative for a second authentication round using the `leftauth2` and `rightauth2` options.

## A.2 Keying Attempts

The **keyingtries** option in the *swanctl.conf* file is a signed integer setting in the *connections* section that determines the number of attempts the system should make to establish an *IKE* connection before giving up.

The general behavior is that, after setting `keyingtries = N`, the daemon will try establishing the connection `N + 1` times, as the first attempt is intrinsically not counted; moreover, by default, the value of the option is set to `1`, while setting it to `0` will prompt the system to continue attempting to establish the connection indefinitely (actually, if a fatal error occurs, the *IKE* daemon does not retry connection regardless of this option).

Note that in the legacy *ipsec.conf* file, a similar and homonym option was present, even if some aspects were treated differently (for instance, the default value was `3` and the unique value `%forever` was required to prompt the daemon to make an indefinite number of attempts).

## A.3 Certification Authorities

The **ca_id** option in the *remote* section of the *swanctl.conf* file is a string type option used for controlling which *CA* certificates are accepted during the authentication process of a connection; it specifies which *Certificate Authority* is considered valid for authenticating the remote peer and can be either the subject name or one of the `subjectAltName` entries in one of the *CAs* of the remote peer trustchain. This option has a similar effect to specifying `cacerts` to force clients under a *CA* to specific connections and does not require the *CA* certificate to be locally available, but it can accept the certificate provided by the remote peer during the *IKE* exchange.

Note that in the legacy *ipsec.conf* file, there was a similar option to specify a *CA* in the trust path to the root authority, which was `rightca`.

## A.4 Public Key Certificates

The **cacert**, **file**, and **handle** options are string type parameters used for specifying how the *CA (Certificate Authority)* certificate should be loaded into the system to trust it for future certificates to be verified. The *file* and *handle* options appear in the *authorities* section and both the *local* and the *remote* subsections for a connection, with the latter having them for the *cert* and the *cacert* sub-subsections. The details are reported here:

- `cacert`: it specifies the path to the *CA* certificate that the daemon should use to verify peer certificates; this path can be either absolute or relative, and, in the latter case, it is assumed to be relative to the `swanctl/x509ca` directory, which is a designated directory for storing *CA* certificates.
- `file`: it specifies an absolute path to the *CA* certificate; this is similar to the *cacert* option, but is completely unrelated to the `swanctl/x509ca` directory.

- `handle`: it specifies the hex-encoded *Cryptographic Key Architecture ID* (which is an attribute specified in the *PKCS#11* standard to uniquely identify cryptographic objects within a *PKCS#11* token) of the *CA* certificate if stored in a cryptographic token, or a handle of the same if stored on a *Trusted Platform Module* respectively.

Note that in the legacy *ipsec.conf* file, only the `cacert` option was present with the same name and purpose, while `file` and `handle` had no direct correspondent.

## A.5 Certificates Validation

*OCSP (Online Certificate Status Protocol)* is a protocol used to check the revocation status of digital certificates in real time. In the context of the *swanctl.conf* file, the *OCSP* settings are handled employing a **ocsp** key to control how *OCSP* is handled at a high level, specifically whether *OCSP* information is included in *IKEv2* payloads as part of the certificate status request and response process, which is related to the *OCSP Content* extension. The values of the key can be:

- `never`: it specifies that *OCSP* requests and responses are never sent nor processed by the peer.
- `request`: the peer can send an *OCSP* status request in a *CERTREQ (OCSP Request)* payload when certificate-based authentication is used, along with the other data during the first messages exchange in the *IKEv2* process (*IKE_SA_INIT* and *IKE_AUTH*); by doing so, the peer identifies a variable number of *OCSP Responders* that are trusted and from which the *Recipient* should demand an *OCSP* response to validate its certificate when replying to the sender.
- `reply`: the peer replies with *OCSP* status response by putting it in a *CERT* payload after having obtained it from one of the trusted *OCSP Responders* indicated by the other peer in a previous request; more specifically, two separate payloads are sent: one *CERT (certificate)* containing the certificate of the peer and the other *CERT (OCSP Response)*, received from the trusted *OCSP Responder*, validating that certificate (note that this is the default setting).
- `both`: the peer sends *OCSP* status requests when certificate-based authentication is used and replies with *OCSP* status responses when a previous request has been received (this merges the `request` and `reply` options).

*OCSP* settings in the legacy *ipsec.conf* file were already adopted, but were managed differently; there was no direct correspondent of the `ocsp` option and the most related one available was `ocspuri`, with the associated `ocsp1` and `ocsp2`, which was used just to specify the *URI (Uniform Resource Identifier)* of an *OCSP* server.

These last options have been ported to the *swanctl.conf* context and have been merged into a single one, renamed as **ocsp_uris** option for the *authorities* section; it is a comma-separated list of *OCSP URIs*, which is left as an empty array in the default behavior.

## A.6 Certificate Revocation Lists

The **crl_uris** option is a setting for the *authorities* section that can be instantiated as a list of comma-separated strings, empty by default. It defines specific locations from where the *CRL (Certificate Revocation List)s* can be downloaded; these data structures contain a list of certificate serial numbers revoked before their scheduled expiry, along with the revocation reason and the revocation date. These *URI*s can point to various protocols such as *LDAP (Lightweight Directory Access Protocol)*, when *CRLs* are managed via a directory service, *HTTP (HyperText Transfer Protocol)*, when *CRLs* are hosted on web servers, and file *URI* when the *CRLs* are stored on the same system or a reachable network share.

Note that in the legacy *ipsec.conf* file, there were similar options used to specify a single *CRL* distribution point instead of a list; these options were `crluri`, `crluri1`, and `crluri2`.

## A.7 Responses to Certificates Revocations

The **revocation** option in the *remote* section of the *swanctl.conf* file is a string type option used to determine how the system handles certificate revocation checks during the peer authentication process; three main possible values indicate the policy used:

- `strict`: it is the most secure and requires that revocation information, such as a *CRL* or an *OCSP* response, must be available and verifiable, otherwise the certificate is treated as if it might be revoked and the authentication attempt fails.
- `ifuri`: it requires revocation checking only if an *OCSP URI* is present in the certificate. If a *URI* is provided but the revocation check fails, the authentication fails; if no *URI* is provided, this lack of information does not cause the authentication to fail.
- `relaxed`: it is the default policy, and the authentication process will only fail if the certificate is explicitly known to be revoked; if no revocation information is available, or if the revocation check cannot be performed, the certificate will still be accepted.

Note that in the legacy *ipsec.conf* file, there was a similar option to specify this kind of policy and had the same modes, even if named differently; the option available was `strictcrlpolicy` and the correspondent values were `yes` for `strict`, `no` for `relaxed` (which was also the default mode), while `ifuri` kept the same name.

## A.8 Certificates References

The **cert_uri_base** option in the *authorities* section of the *swanctl.conf* file is part of *Hash and URL*, an optimization feature supported by *IKEv2*. This method allows for an efficient certificate exchange by sending just a *URI* that points to the certificate rather than the complete certificate data, heavily reducing the amount of data transmitted during the negotiation; the *URI* is constructed by appending the *SHA1* hash of the certificate's details encoded form to the base *URI* specified in this option. For instance, if the option is set as `cert_uri_base = http://example.com/certs/` and the hash of a certificate is `abc123`, the resulting *URI* sent during *IKE* negotiation would be `http://example.com/certs/abc123`, which the receiver will use to fetch the actual certificate directly using standard web protocols.

Note that in the legacy *ipsec.conf* file, there was a similar option, which was `certuribase` and had the same purpose.

## A.9 Hardware Offload

The **hw_offload** option in the *swanctl.conf* file is a string type option used to specify whether *IPsec* offloading to hardware is enabled for a specific *CHILD_SA*; hardware offloading can help improve the performance of *IPsec* operations by allowing specific hardware to handle certain cryptographic tasks that would otherwise be performed via software. The possible values that can be assigned to the `hw_offload` option are described here:

- `yes`: it enables hardware offloading using the default method supported by the system, either `crypto` or `packet` offloading; if no specific hardware offloading is supported, the installation will fail.
- `no`: it disables hardware offloading, and all *IPsec* related processing will be handled through software procedures; note that this is the default behavior if it is not specified differently.
- `auto`: it instructs to attempt enabling full packet offloading or `crypto` offloading if the hardware supports one of them; if neither is supported, the installation will not fail but will revert to software processing.
- `crypto`: it specifies that only cryptographic operations should be offloaded to hardware, so encryption and decryption tasks are handled by the hardware, but other packet processing, such as encapsulation and decapsulation, is done via software. If this offloading type is selected but not supported, the installation of the *CHILD_SA* will fail.

- `packet`: it enables offloading entire packet processing to hardware, not just cryptographic tasks but also other related *IPsec* operations. On Linux systems, this setting also includes offloading of policies. If this offloading type is selected but not supported, the installation of the *CHILD_SA* will fail.

Note that in the legacy *ipsec.conf* file, there was no support for similar options, as hardware offload was not supported.

## A.10 Hardware Modules

The **module** option in the *swanctl.conf* file is related to the integration of *PKCS#11* modules, which is a standard that defines a platform-independent *API* to cryptographic tokens such as *HSM (Hardware Security Module)s*; it specifies the name of the *PKCS#11* module that the system should load to interact with cryptographic hardware. By default, this option, which is a string type parameter, is left empty and, when specified, instructs the system to load the corresponding *PKCS#11* module with the provided name. This option appears in the *authorities* section and in both the *local* and the *remote* subsections for a connection, with the latter having them for the *cert* and for the *cacert* sub-subsections.

Note that in the legacy *ipsec.conf* file, there was the possibility to specify similarly a *PKCS#11* module to rely on. It was to be indicated in the `leftcert` and `rightcert` options through a specific syntax.

## A.11 Hardware Modules Slots

The **slot** option in the *swanctl.conf* file is related to the use of *PKCS#11* cryptographic tokens, like discussed for the `module` option. It is a signed integer type parameter used to specify which slot of the cryptographic module should be accessed to retrieve the necessary cryptographic information; each slot of the *PKCS#11* module can contain one or more tokens that are useful for cryptographic tasks; the `slot` option specifies which of these slots should be used to access the necessary cryptographic tokens. It appears in the *authorities* section and in both the *local* and the *remote* subsections for a connection, with the latter having them both for the *cert* and the *cacert* sub-subsections.

Note that in the legacy *ipsec.conf* file, there was the possibility to specify similarly a *PKCS#11* module's slot to retrieve cryptographic information, and, similarly to the `module` option, it was to be indicated in the `leftcert` and `rightcert` options through a specific syntax.

## A.12 Interface Identifiers

In the context of the new *swanctl.conf* file, the **if_id_in** and **if_id_out** options have been introduced and their purpose is to configure *XFRM Interface Identifiers* for respectively inbound and outbound *IPsec* traffic through *IPsec* tunnels.

- `if_id_in`: it specifies the *Interface Identifier* expected for the incoming *IPsec* packets of a specific inbound *SA*.
- `if_id_out`: it specifies the *Interface Identifier* to be used for the outgoing *IPsec* packets of a specific outbound *SA*.

Both these options can appear at *connection* level or *child* level; in the former, the *Interface Identifier* is set for the *IKE_SA* and gets inherited by the *CHILD_SAs* unless overridden in its scope. Conversely, in the latter, the *Interface Identifier* is explicitly set for the *CHILD_SA*.

By default, the *Interface Identifiers* are set to `0`, meaning identifying an interface is not required. Moreover, if it is desired to set a single interface for inbound and outbound traffic, other than assigning the same ID to both options, the value `%unique` can be assigned to generate a unique *Interface Identifier* for each *CHILD_SA*, valid for both directions; on the other hand, if it is desired to use separate interfaces for inbound and outbound traffic, other than assigning different IDs to the 2 options, the value `%uniquedir` can be assigned to generate a unique *Interface Identifier* for each *CHILD_SA* and direction.

It is to be noted that, in the legacy *ipsec.conf* configuration, there was no direct equivalent to `if_id_in` and `if_id_out`, as the general approach was to rely on the combination of *IP* addresses, subnets, and *SPI (Security Parameter Index)s* to match and manage *IPsec* tunnels.

## A.13 Interface Names

The **interface** option in the *swanctl.conf* file is a string parameter that belongs to the *children* section and which defines an optional interface name for outbound *IPsec* policies. By default, this option is set as `null`, meaning that outbound packets are processed on potentially any interface; as a consequence, the use of the *interface* option for a *CHILD_SA* is to restrict a policy to a single interface for specific use cases.

From the official *strongSwan* documentation, an example of the usage for this option can be found at `https://strongswan.org/testing/testresults/ikev2/shunt-manual-prio/moon.swanctl.conf`, in which a default *drop* policy is installed on the external interface *eth0*.

## A.14 Security Labels

The **label** and the **set_label** options have been introduced in the *swanctl.conf* file to manage security labels for *IKEv2* connections; a security label can be seen as a tag that the operating system uses to apply specific security policies to the packets tagged with that label, allowing the integration of security systems to enforce connections, like *SELinux* (a security architecture for Linux systems that allows firm control over who can access system resources, more details can be found at `https://www.redhat.com/en/topics/linux/what-is-selinux`). The details of these options are presented here:

- `label`: it is used to specify a security label, such as an *SELinux* context; it is a string type option that might look like `system_u:object_r:ipsec_spd_t:s0` if, for instance, the *selinux* plugin has been enabled.

- `label_mode`: it is used to define how the security label configured in the `label` option is used by *strongSwan*. It can be set with 3 possible modes:

  - `simple`: the label is used strictly as provided in the `label` option, functioning as an additional identifier during the negotiation of *CHILD_SAs* and the selection of configuration profiles; labels applied through this mode are not involved in kernel-level operations, meaning that they are not used to influence packet handling by the operating system itself, and labels received from a peer with which a connection is being established have to match precisely the ones configured locally for the connection to be successful.

  - `selinux`: it is used to indicate a generic *SELinux* context as a label and is selectable only if *SELinux* is available and enabled on the system. In this case, the label is installed in the *trap* policies (meaning that the option `start_action = trap` is usually associated with this one), which are used to capture packets that do not yet have a matching *SA* and require one to be negotiated to employ an *acquire* event that is triggered; if a connection is initiated directly, without an *acquire* event, a childless *IKE_SA* is established. Trap policies are installed at both ends of the connection to ensure that the necessary *CHILD_SAs* are established when required. One difference between this mode and the `simple` mode is that labels received from peers are accepted if they match the locally configured label based on `association:polmatch`. This mechanism evaluates whether two *SELinux* labels are compatible.

  - `system`: it is a default value that automatically selects `selinux` if supported or `simple` otherwise.

Note that in the legacy *ipsec.conf* file, there was no support for similar options, as security label management was not supported.

## A.15 Policy Priorities

The **priority** option in the *swanctl.conf* file helps manage *IPsec* policies; it is a *children* section option that can assume an unsigned integer value, and that allows to set a fixed priority for *IPsec* policies. *IPsec* policies determine the handling of traffic based on criteria such as source and destination *IP* addresses and ports, and the priority of a policy is used to determine which one should be applied when multiple policies match the same traffic pattern.

By default, the priority value of a policy is set to `0` so that the priority of a policy is dynamically calculated based on the specificity of the *traffic selectors*; for instance, a policy that specifies a single *IP* address as a selector is given a higher priority (due to being more specific) compared to a policy that defines an entire subnet; as a consequence, setting a fixed priority overrides this dynamic calculation, allowing a policy to be treated as more or less important regardless of its selectors' characteristics. This option is beneficial for defining security policies that should always take precedence, such as dropping potentially harmful traffic before it can reach sensitive parts of a network. When multiple policies might match the same traffic, administrators can use fixed priorities to ensure that the intended policy always takes precedence.

The legacy *ipsec.conf* file had no direct equivalent of this `priority` option to allow fixed priorities; it relied substantially on automatically calculated priorities based on the specificity of the traffic selectors, which is a similar behavior to the default one in *swanctl.conf*.

## A.16 Inner-Outer IP Header Parameters

In the *swanctl.conf* file, three options have been introduced to indicate whether to copy some specific information from the original/inner *IP* header to the new/outer *IP* header when operating in *IPsec tunnel* mode. These options are **copy_df**, **copy_ecn** and **copy_dscp**, and the details are here described:

- `copy_df`: a boolean type option that is used to control whether the *Don't Fragment* bit of the inner *IP* header should be copied to the outer *IP* header in tunnel mode. The *Don't Fragment* bit, when set, instructs routers not to fragment the packet, so if it is set and the packet is larger than the *MTU (Maximum Transmission Unit)* of any segment along the path, it will be dropped instead of fragmented. By default, this option is set to `true`. It is useful for disabling *Path MTU Discovery*, a mechanism to determine the maximum packet size that can travel safely through the network without needing fragmentation; if the *Don't Fragment* bit is set, copying it to the outer header maintains the original packet's intent of not being fragmented during the path.

- `copy_ecn`: a boolean type option that is used to allow the *ECN (Explicit Congestion Notification)* bits from the inner *IP* header to be copied to the outer *IP* header. *ECN* is a feature available in modern networks to reduce network congestion without dropping packets, as the basic idea is that routers supporting it can mark packets instead of dropping them, signaling to endpoints to reduce their transmission rate; by default, this option is set to `true` and is useful to maintain the original packet's intent of signaling congestion through this mechanism.
- `copy_dscp`: a string type option that is used to control whether the *DSCP (Differentiated Services Code Point)* field, which is used for *QoS (Quality of Service)* tagging in *IP* packets, is copied from the inner *IP* header to the outer *IP* header or vice versa. This option can be set to:
  - `out`: the *DSCP* value is copied from the inner *IP* header to the outer *IP* header when encapsulating the packet and guarantees that the priority assigned to the original traffic is maintained as the packet travels through the tunnel.
  - `in`: the *DSCP* value is copied from the outer *IP* header to the inner *IP* header when decapsulating the packet and leads to the network's influence on the *QoS* for the traffic to be maintained once it exits the tunnel.
  - `yes`: the *DSCP* values are copied in both directions, when encapsulating and decapsulating, ensuring that *QoS* settings are preserved throughout the entire path of the packet.
  - `no`: the copy of the *DSCP* field is disabled both during encapsulation and decapsulation, ensuring that no *QoS* settings from the inner packets to the outer packet or vice versa are carried.

  The default setting for this option is `out` because adopting `yes` or `no` could allow an attacker to affect the traffic at the receiver site.

Note that these behaviors are not supported by all kernel interfaces and there were no similar options in the legacy *ipsec.conf* file.

## A.17 Childless Security Associations

During the normal execution of the first exchanges in the *IKEv2* protocol, after the initialization of the connection and the authentication of the parties, the first *CHILD_SA* is created, attaching its details in the *IKE_AUTH* request/response messages.

The **childless** capability in strongSwan is a string type parameter in the *connections* section used to manage how *IKEv2* connections handle the initiation of the first *CHILD_SA*, allowing for a more flexible initiation process by potentially delaying its creation until after the *IKE_SA* has been established, through a separate *CREATE_CHILD_SA* exchange; there are four possible settings:

- `allow`: as a *Responder*, the device will accept childless *IKE_SAs* (indicated by a special notification in the *IKE_SA_INIT* response). As an *Initiator*, the device will continue to initiate *IKE_SAs* with the first *CHILD_SA* included during the *IKE_AUTH* exchange, unless it is explicitly requested to perform the initiation without any children; if instructed to do so, the process will fail if the *Responder* does not support or has disabled childless initiations (note that this is the default option).
- `prefer`: as a *Responder*, the behavior is the same of the `allow` option, while, as an *Initiator*, if the *Responder* supports childless *IKE_SAs*, the device will prefer to initiate an *IKE_SA* without any *CHILD_SA* during the initial exchange.
- `force`: forces the device to only accept or initiate childless connections, regardless of whether it is an *Initiator* or a *Responder*; this setting mandates that no *CHILD_SA* is created during the initial *IKE_AUTH* phase, ensuring that all *CHILD_SAs* are initiated separately via a *CREATE_CHILD_SA* exchange.
- `never`: disables support for childless *IKE_SAs* completely, both as an *Initiator* and as a *Responder*; this setting ensures that any attempt to initiate an *IKE_SA* without a *CHILD_SA* will be rejected.

This feature does not correspond directly in the legacy *ipsec.conf* file, as all connections inherently assumed the negotiation of at least one *CHILD_SA*.

## A.18 One User - Multiple Connections

The **unique** option in the *swanctl.conf* file is a string setting in the *connections* section that is used to control the behavior of handling multiple concurrent connections from the same user or identity, specifying how uniqueness should be handled.

The system determines connections' uniqueness relying on the remote *IKE* identity during the *IKE* negotiation, unless *EAP* or *XAuth* is involved, in which case the *EAP*-Identity or *XAuth* username replaces the *IKE* identity for this purpose. For the initiating party, this setting influences whether an *INITIAL_CONTACT* notification is sent during the *IKE_AUTH* phase if no existing connection with the remote peer's identity is found; sending this notification can inform the remote peer to replace or remove any existing connections under their uniqueness policy. The possible values for this option are **never**, **no**, **keep** and **replace**, and the details of each are here described:

- `never`: it is the most permissive value, used to specify that no uniqueness policy is enforced; as a consequence, multiple connections from the same identity are allowed, regardless of whether the *INITIAL_CONTACT* notification is included by the other peer. From an initiator's point of view, this is the only option that prevents sending the *INITIAL_CONTACT* notification during the *IKE_AUTH* phase.

- `no`: it is the default setting, used to specify that existing connections for the same identity are replaced if a new one includes an *INITIAL_CONTACT* notification; from an initiator's point of view, an *INITIAL_CONTACT* notification is sent during the *IKE_AUTH* phase.
- `keep`: it is used to reject new connection attempts if there is already an active connection from the same identity, keeping active only the original one for a specific user; from an initiator's point of view, an *INITIAL_CONTACT* notification is sent during the *IKE_AUTH* phase.
- `replace`: it is used so that any existing connection from the same identity is deleted when a new connection is established, keeping active only the most recent one for a specific user; from an initiator's point of view, an *INITIAL_CONTACT* notification is sent during the *IKE_AUTH* phase.

Note that in the legacy *ipsec.conf* file, a similar option with homonym values was present and was defined as `uniqueids`; the significant difference concerning the new context is that `replace` was the default value instead of `no`.

## A.19 Mediation Servers

The mediation capabilities provided by *strongSwan* are part of its support for the *IKEv2 Mediation Extension*, which facilitates the setup of *IKEv2* connections through a mediation server; this can be useful in scenarios where direct connections between peers are not possible due to *NAT (Network Address Translation)* traversal issues or firewall restrictions. The capabilities introduced as part of the new **swanctl.conf** context are:

- `mediation`: it specifies whether a connection acts as a mediation connection for others; if this is set to `yes`, the connection's primary role is to facilitate the establishment of other connections, rather than to secure data itself; this is why no *CHILD_SA* is created (note that, by default, this option is set to `no`).
- `mediation_by`: it is a string specifying another connection that should be used as *mediator* to establish the current one; it is to be noted that the referred connection should have the `mediation` option set to `yes` for this procedure to work.
- `mediation_peer`: it is the *IKE* identity the other end of this connection uses as its local identity on its connection to the mediation server; more specifically, a peer *P1* with a defined mediation connection with a mediation server *MED*, for instance, *P1toMED*, and a connection with another peer *P2*, for example *P1toP2*, can set this option in the *P1toP2* connection to specify the identity that *P2* uses for its connection with *MED*, for instance *P2toMED*. This option is relevant only on connections that set the `mediated_by` option, and if not specified, the default identity used is the one from the first authentication round.

In the legacy *ipsec.conf* file, these functionalities were already present with similar names and functionalities (`mediation`, `mediation_by`, `me_peerid`).

## A.20 Post-Quantum Cryptography

The options **ppk_id** and **ppk_required**, which relate to PPK (Post-Quantum Pre-Shared Key)s, are new options introduced with the *swanctl.conf* file and do not have direct correspondents in the legacy *ipsec.conf* context.

*PPKs* are part of an effort to enhance the security of *IKEv2*; *RFC 8784* addresses the potential threat posed by quantum computers to current cryptographic algorithms, particularly those used in *IKEv2*; traditional pre-shared keys can be vulnerable to quantum computer attacks and, to mitigate this, a method has been introduced to strengthen *PSKs* with an additional layer of security, which are the *PPKs*.

A *PPK* is not intended to replace the original *PSK* but rather to enhance it, and the basic idea is to apply a secret transformation to the original *PSK* before it is used in the *IKEv2* key exchange process; this transformation makes the derived keys more resistant to decryption by quantum computers.

Here are presented the details about the new options:

- `ppk_id`: it is a string identifier for the *PPK*; when a connection is initiated and a *PPK* is required, it indicates to the device which one to use; it is necessary if there are multiple *PPKs* configured on a server, each potentially used for different connections (note that, by default, this option is `null`).
- `ppk_required`: this boolean setting specifies whether the use of a *PPK* is mandatory for establishing the connection; if set to `yes`, the connection will only be established if both sides support a *PPK* while, if set to `no` (which is the default setting), the connection proceeds under a regular *PSK*.

Note that the details about the *PPKs*, in the *swanctl.conf* file, are defined in a specific *secrets* section (`secrets.ppk<suffix>`).

# Bibliography


[1] Aurelio Cirella. *An abstract model of NSF capabilities for the automated security management in Software Networks*. 2021.

[2] Sheila Frankel and Suresh Krishnan. *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap*. RFC 6071. Feb. 2011. DOI: 10.17487/RFC6071. URL: https://www.rfc-editor.org/info/rfc8329.

[3] GeeksforGeeks. *Unified Modeling Language (UML) | An Introduction*. 2019. URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction.

[4] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 2013. URL: https://www.w3.org/TR/xml/.

[5] World Wide Web Consortium. *W3C XML Schema Definition Language (XSD) 1.1*. 2012. URL: https://www.w3.org/TR/xmlschema11-1/.

[6] Object Management Group. *XML Metadata Interchange (XMI) Specification*. 2015. URL: https://www.omg.org/spec/XMI/2.5.1/PDF.

[7] The strongSwan Team and individual contributors. *strongSwan*. 2023. URL: https://docs.strongswan.org/docs/5.9/howtos/introduction.html.

[8] Diego Lopez, Edward Lopez, Linda Dunbar, John Strassner, and Rakesh Kumar. *Framework for Interface to Network Security Functions*. RFC 8329. Feb. 2018. DOI: 10.17487/RFC8329. URL: https://www.rfc-editor.org/info/rfc8329.

[9] E. Haleplidis, K. Pentikousis, S. Denazis, J. Hadi Salim, D. Meyer, and O. Koufopavlou. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426. Jan. 2015. DOI: 10.17487/RFC7426. URL: https://www.rfc-editor.org/info/rfc7426.

[10] Cataldo Basile, Daniele Canavese, Leonardo Regano, Ignazio Pedone, and Antonio Lioy. «A model of capabilities of Network Security Functions». In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)* (Aug. 2022).

[11] Dario Marchitelli. *An abstract model of NSF capabilities for the automated security management in Software Networks*. 2024.