POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Models and strategies for automated security policy refinement

Supervisors:

Prof. Cataldo Basile

Dott. Francesco Settanni

Candidate:

Davide Belluardo

Academic Year 2023/2024
Torino

Abstract

In the rapidly evolving cybersecurity domain, refining high-level security policies is essential to effectively manage network threats' increasing complexity and diversity. This thesis builds on prior research that established a sophisticated refinement process to transform high-level policy directives into technical configurations that can be subsequently applied to network devices. This approach starts with parsing high-level specifications and network topology data to derive enforceable rules that align with the underlying network architecture and security requirements. Central to this process is using a Domain-Specific Language to craft expert systems called CLIPS, which enrich the achieved policy interpretations by extracting essential details from the abstract policy definitions.

This thesis work introduced several enhancements that aim to optimize the policy refinement process for complex network systems by incorporating advanced methodologies and tools to streamline the alignment of security measures with network architectures and ensure that the configurations are functional and pertinent to the specific operational environments. For instance, it supports the selection of combinations of NSFs, instead of a single NSF, to fulfil the entire set of needed capabilities and provide a more granular selection of security controls. It also enables smart updates of technical configurations when high-level policies are updated, reducing the need for complete reconfiguration from scratch. A significant advancement involves introducing a standardized TOSCA-based model for network topology description. This model provides a structured representation of network layouts and device features, which is crucial for properly applying security policies.

The effectiveness of the proposed solutions is demonstrated through rigorous testing, confirming their ability to generate accurate and up-to-date configurations and concrete low-level policies, which can be enforced over various network scenarios. This thesis extends the use cases of the existing policy refinement system, providing a more scalable and flexible solution and building the basis for future developments in automated security policy management.

Acknowledgements

As my time at the Politecnico di Torino comes to the end, I'm taking a moment to reflect on the journey... It's been a period of immense growth and learning, full of challenges that have inspired me to reach my full potential.

I want to express my sincere gratitude to my academic supervisors, Professor Cataldo Basile and Dott. Francesco Settanni, for their guidance and consistent support throughout my thesis work. Your practical advice and timely feedback were significant to approach the complexities of my research with a clear and determined mindset. Thank you for your patience and profound dedication.

A deep sense of appreciation goes to my family and friends for constantly believing in me. You have been my strength, standing beside me in the hardest days and celebrating the moments of success. Your encouragement and presence have meant more to me than words can express.

Finally, I want to acknowledge all the people I've met along the way. From collaborative projects and exams we faced together to the countless moments shared around university spaces, each of you has left an indelible mark on this path.

Thank you all for enriching and being part of this chapter of my life.

Table of Contents

List of Figures

Acronyms

API

Application Programming Interface

CFFI

C Foreign Function Interface

CIDR

Classless Inter-Domain Routing

CLIPS

C Language Integrated Production System

CNCF

Cloud Native Computing Foundation

CRD

Custom Resource Definitions

CURL

Client for URL

DID

Distributed ID

DMTF

Distributed Management Task Force

DNS

Domain Name System

EDC

Enforcement and Dynamic Configuration

GUI	Graphical User Interface
HSPL	High-Level Security Policy Language
I2NSF	Interface to Network Security Functions
IaaS	Infrastructure as a Service
ICT	Information and Communications Technologies
IETF	Internet Engineering Task Force
IP	Internet Protocol
IT	Information Technology
JSON	JavaScript Object Notation
K8s	Kubernetes
MAC	Media Access Control
MSPL	Medium-Level Security Policy Language
NSF	Network Security Function
OASIS	Organization for the Advancement of Structured Information Standards

PaaS

Platform as a Service

PDP

Policy Decision Point

PEP

Policy Enforcement Point

PF

Packet Filter

REST

Representational State Transfer

SaaS

Software as a Service

SACM

Security Assurance and Certification Management

SC

Security Control

SCM

Security Capability Model

TOSCA

Topology and Orchestration Specification for Cloud Applications

VPN

Virtual Private Network

WID

Wallet ID

XML

Extensible Markup Language

YAML

YAML Ain’t Markup Language

Chapter 1

Introduction

In an era where technological advancements are redesigning every aspect of society, the field of cybersecurity emerges as one of the pillars for protecting digital infrastructures. Modern networks support essential services, from finance and healthcare to energy and communication, making them prime targets for increasingly sophisticated cyber threats. The rapid evolution of attack strategies, such as ransomware, hard persistent threats, and zero-day exploits, exposes the limitations of traditional security measures, calling for more advanced, scalable, and adaptive defenses.

Among the diverse domains of cybersecurity, network security plays a core role, serving as the first line of defense against unauthorized access, data breaches, and service disruptions. However, safeguarding network infrastructures has become a tough task. The complexity of modern networks, characterized by the convergence of on-premises systems with cloud-native environments and the growing reliance on distributed architectures, presents significant purposes to achieve. These developments demand innovative approaches, so that they can dynamically adapt to evolving requirements while maintaining robust protection.

The following sections offer an overview of the *context*, *motivations*, and *objectives* behind the research presented in this thesis, along with the *organization of contents* across the chapters.

1.1 Context and Motivations

The illustrated landscape necessitates tools and frameworks capable of dealing with ever-evolving threats, while accommodating the complexity of today's network environments. High-Level Security Policies (*HSPLs*) are fundamental in defining abstract security intentions, yet their effective translation into concrete configurations remains a significant challenge. As networks expand in scale and encompass a diverse multiplicity of devices and architectures, traditional policy-based management sometimes struggles to narrow the distance between high-level objectives and the feasibility of device-level configurations.

This thesis operates in the context of *automated refinement* frameworks for HSPLs, focusing on their critical role in transforming abstract security requirements into enforceable rules. They have become indispensable for recent security solutions, allowing organizations to maintain alignment between strategic policies and the operational realities of their network infrastructures. However, as networks grow in complexity, traditional approaches to policy refinement could encounter limitations, involving inefficiencies in handling dynamic architectures and adapting to evolving security demands.

The starting point of this research is an existing framework that successfully demonstrated the ability to automate the transformation of HSPLs into enforceable configurations for network security devices. Despite its central contributions in illustrating the potential of automating a traditionally manual process, the framework revealed certain limitations, highlighting the necessity of advancements to address these flaws. The need is further amplified by the growing demand for solutions that integrate into such heterogeneous ecosystems:

- *scalability*: as networks expand in size and scope, the framework should process a growing number of policies, devices, and users. Therefore, minimizing manual effort and operational inefficiencies is imperative, particularly in environments with high requests for responsiveness;
- *adaptability*: ongoing adaption to new network entities or evolving policy requirements evidences the need for a paradigm shift in how security policies are managed and implemented;
- *interoperability*: adhering to current standards is one of the main concerns for tools that operate across diverse environments, enabling hybrid and contemporary network configuration setups.

Overcoming these challenges necessitates novel approaches suited to the forward demands of modern network security. This thesis proposes targeted improvements to refine processes and better accommodate the needs of expanding, evolving, and interconnected network environments to face the imperfection of the available framework.

1.2 Objectives

The primary objective of this thesis is to enhance the automated refinement of *High-Level Security Policies* into applicable configurations. This will drive the process towards more scalable and adaptive modes, by integrating innovative standards and strategies to provide more systematic access to policy refinement. To achieve this, the research focuses on the following key goals:

- *defining formalized methodologies*: develop a structured and standardized representation of network components and their relationships, enabling validation and relative data retrieval, and ensuring compatibility with cloud-native platforms;

- *extending the selection of security controls*: formulate sophisticated approaches to identify and combine security controls, splitting policy conditions when necessary to allow comprehensive enforcement where the traditional method fails;
- *upgrade policy revisions*: design processes to handle configuration updates efficiently, minimizing redundancies, eliminating the need for full resets, and maintaining synchronization with current network conditions and policies;
- *boosting adaptability and scalability*: improve the framework's responsiveness to changes in network structures and security demands, keeping high performance while reducing human intervention.

By addressing these intents, this thesis seeks to elevate the refinement framework's ability to manage the complexity and dynamism of modern networks within security environments. Through the solutions developed, the work aims to remedy existing tool's limits, establishing a more robust and efficient system for policy management while setting the prerequisites for future innovations in automated network security. The chapters that follow will present the research plans, procedures, and results.

1.3 Chapters Overview

This thesis structure is organized as follows:

- **Chapter 2**: provides the theoretical background on HSPLs, policy refinement, and the technologies supporting network security explored in this work;
- **Chapter 3**: reviews the literature and related works that have been considered and analyzed for their approaches concerning the automated policy refinement;
- **Chapter 4**: defines the problem statement and identifies some key limitations of the existing framework;
- **Chapter 5**: details the design of the proposed advancements, addressing the exposed challenges;
- **Chapter 6**: describes the implementation of the presented solutions within the refinement framework;
- **Chapter 7**: validates the contributions through targeted testing and practical scenarios;
- **Chapter 8**: draws conclusions, discusses the findings, and outlines future research directions.

Chapter 2

Background

This chapter will present the key concepts, technologies, and languages needed for understanding the context of the advancements carried out in this thesis. The main notions concern the ability to define, analyze, and process security policies in networked systems, exploring the roles of policy frameworks, policy-based management, and expert system tools. Further knowledge deals with proper formats and models involved in defining network layouts and other strategies supporting security policy processing.

2.1 Security Policy

Security policies are fundamental mechanisms designed to specify and enforce security measures across systems and networks. Their main goal is to protect sensitive information and manage access by defining acceptable user actions, specifying conditions for access, and restricting unauthorized activities. In networked environments, security policies prevent malicious access and ensure that users comply with defined security standards, thus safeguarding systems from threats and vulnerabilities. These policies also facilitate effective security management, as they can be adapted to fit various levels of user expertise, from non-technical users to network administrators. To meet the needs of diverse users, security policy languages are often divided into different levels of abstraction, with *High-Level Security Policy Languages* (*HSPL*) providing a simplified interface, while *Medium-Level Security Policy Languages* (*MSPL*) focus on translating these high-level directives into actual configurations suitable for technical deployment [1].

2.1.1 HSPL

The **High-Level Security Policy Language** (**HSPL**) is specifically designed to allow non-technical users to define and manage security policies without needing deep technical knowledge. HSPL enables users to outline protection requirements intuitively, using simplified constructs that approximate natural language. For instance, users can specify restrictions like "*block internet access during work hours*" or "*allow VPN access only on* *certain networks*". By providing predefined policy statements and auto-completion tools, HSPL offers an approach that reduces the complexity typically associated with security configurations [1].

HSPL is characterized by the following core features:

- *simplicity*: users can easily create policies through familiar terms and phrases, e.g. allowing or blocking specific resources, without dealing with technical syntax;
- *flexibility*: the language accommodates various types of security policies and allows users to specify specific conditions, e.g. time constraints or resource limitations;
- *extensibility*: HSPL is designed to evolve, enabling the addition of new policy types and conditions without altering the core structure.

An HSPL statement generally includes the following components:

- `subject`: the user or entity performing an action, such as an employee or a system;
- `action`: the desired operation;
- `object`: the target resource or entity, such as a type of network traffic or a specific application;
- `optional condition`: additional parameters, like time or content type, that further define the scope and application of the action.

With these components, HSPL enables users to set high-level security requirements effectively, thus enhancing network protection without requiring technical expertise.

2.1.2 Policy-based Management

Policy-based Management offers a structured approach to simplify network administration, automating configuration across network infrastructure. As networks grow more complex, configuring devices individually becomes impractical, particularly with the rise of specialized protocols and standards. By centralizing control, policy-based management allows administrators to define high-level directives that automatically propagate to network components, improving efficiency and reducing configuration errors [2].

As shown in Figure 2.1, this approach distinguishes between two levels of policies: *business-level* and *technology-level*. Business-level policies align with organizational goals and operational needs, allowing administrators to define policies in terms of service requirements or security standards without technical specifics. These are then translated into technology-level policies, specifying the technical configurations necessary for implementation. This approach enables administrators to focus on strategy, while automated tools handle low-level details.

The key elements of the policy framework standardized by the *Internet Engineering Task Force* (*IETF*), together with the *Distributed Management Task Force* (*DMTF*), include

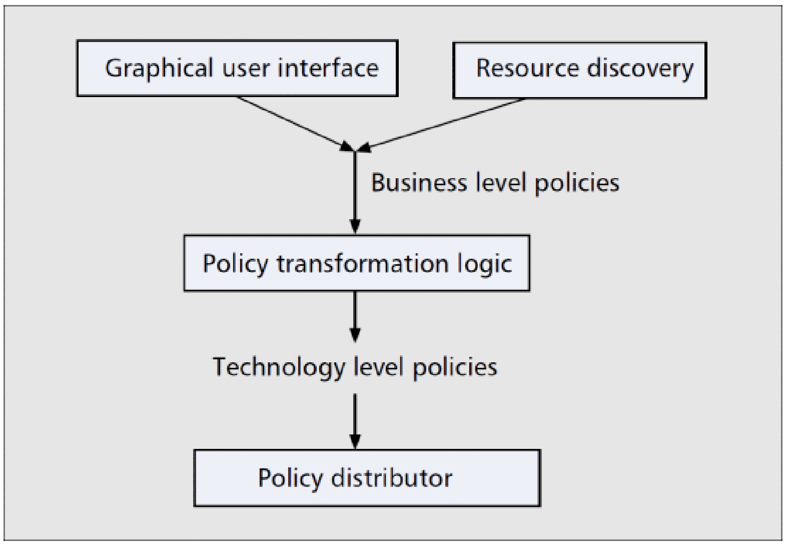


Figure 2.1: Generic policy management tool

the *policy management tool*, *policy repository*, *policy decision point* (*PDP*) and *policy enforcement point* (*PEP*), as illustrated in Figure 2.2. The management tool provides the interface for defining policies, which are stored in the repository. The PDP interprets these policies and sends configurations to PEPs, which enforce them directly on network devices. IETF standards often support interoperability, ensuring consistent policy application across devices from various vendors.

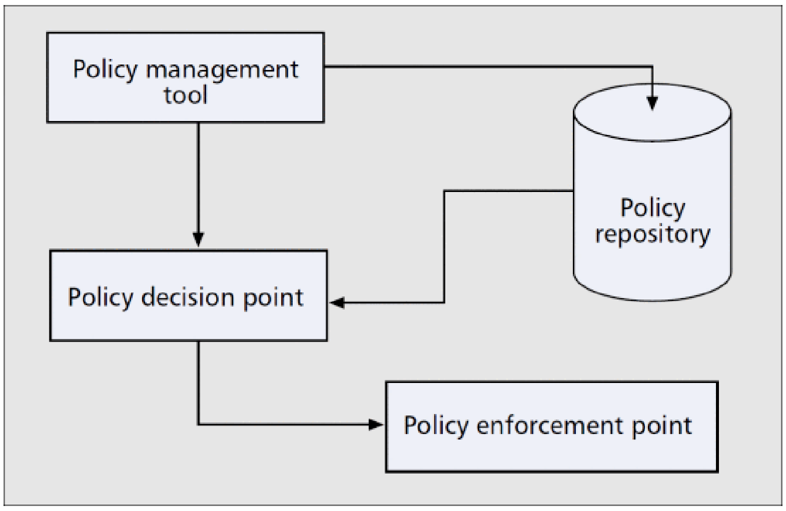


Figure 2.2: IETF/DMTF policy framework

The benefits of policy-based management are significant. Centralized control reduces manual effort and minimizes inconsistencies. By abstracting policies at the business level, the approach is accessible to administrators with variable technical expertise, ensuring alignment between IT operations and business goals. Policy-based frameworks often include validation mechanisms to prevent conflicts and ensure that policies are consistent and feasible, maintaining a secure and efficient network.

2.1.3 Security Policy Refinement

Security Policy Refinement represents the process of translating high-level security policies into enforceable rules within the actual environment. Security policies, typically formulated in abstract terms such as subject and object roles in the system, with permitted or forbidden actions, serve as the outline for designing and enforcing security controls on system components. The process of refining these abstract concepts into concrete security mechanisms without losing their intended purpose is critical. The main focus is ensuring a close alignment between policy intentions and their practical implementations [3].

Security policies are often articulated in a high-level language that emphasizes the protection goals of a system, such as safeguarding against unauthorized access or ensuring data privacy. These policies may define security in terms of broad attributes like *confidentiality*, *integrity*, *availability*, or *privacy.* Common frameworks and models used to state these policies are instrumental in delineating *authorized* versus *unauthorized* actions or *secure* versus *insecure* states within a system.

For systems requiring high assurance, such as those handling sensitive or legally protected data, it is imperative not only to define security policies but also to ensure these policies are correctly enforced. For this reason, policy refinement is crucial for preventing potential security breaches that could arise from misinterpretations or implementation flaws. The refinement process involves mapping from *high-level* security specifications to *medium-level* policies and then down to the actionable controls within the system's architecture. This includes the generation of specific *low-level* configurations starting from general policy statements (see Figure 2.3).

One of the major challenges in policy refinement is ensuring that the refined policies are comprehensive and enforceable without introducing vulnerabilities. The complexity of modern systems often leads to a situation where high-level policy goals do not directly correspond to a single mechanism, but rather to a combination of interleaved controls that operate at different layers of the system's architecture. Another need is to adapt to changing threats and technological demands for ongoing refinement of policies, so as to accommodate new security requirements.

Ensuring that security policies are not only properly translated but also validated to safeguard the system's operations remains essential, emphasizing the necessity of strong verification processes to align practical implementations with their original security intents.

Keeping the core of developing more robust models for policy refinement will be necessary to handle the increasing complexity of systems and the diversification of threats, ensuring that security policies remain effective and enforceable over time. This will include exploring automated refinement tools that can support the precise implementation of complex policies across diverse system components.

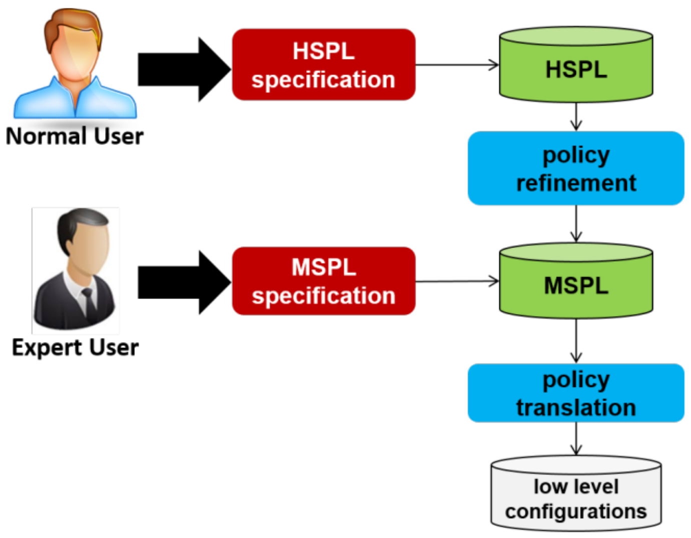


Figure 2.3: From HSPLs to security control configurations

This structured approach to policy refinement ensures that security policies are not only theoretical frameworks but also enforceable measures to protect systems in the real world.

2.2 CLIPS

CLIPS, an acronym for **C Language Integrated Production System**, is an *expert system* development tool that was originally designed by the *Software Technology Branch* at *NASA's Johnson Space Center* in 1985 and continued until 1996. Developed to facilitate the modeling of human expertise, CLIPS has evolved to be a widely adopted tool across various domains, from aerospace to knowledge management [4].

The motivation behind the creation of CLIPS was NASA's need for a versatile and efficient tool to construct expert systems capable of simulating human decision-making processes. Unlike its predecessors, CLIPS was built to be highly portable, written in C to leverage the language's speed and compatibility with existing systems and platforms. This initial design choice has made it integral in projects that require a robust integration of decision-making logic with operational systems.

CLIPS mainly stands out for its ability to support multiple programming paradigms within a single environment:

- *Rule-Based Programming*: at its core, CLIPS operates as a rule-based system where rules determine the logic according to which decisions are made. This is mostly used for encoding heuristic knowledge based on real-world experience;

- *Procedural Programming*: through def-functions and generic functions, CLIPS supports procedural programming, allowing it to handle complex tasks that are better managed through procedural code rather than rules;
- *Object-Oriented Programming*: supporting modern programming practices, CLIPS incorporates object-oriented features such as classes, inheritance, and polymorphism. This allows users to encapsulate data and operations within objects, improving the modularity and reusability of code.

Central to CLIPS's functionality is its *inference engine*, which utilizes the **RETE algorithm**, an efficient pattern-matching algorithm for implementing rule-based systems [5]. The RETE algorithm raises the performance of the expert system by minimizing the overhead of recalculating the conditions of **rules** after a change in **facts**. In CLIPS, rules are written as if-then statements, where the "*if*" part specifies a pattern to match against a set of facts or data in the working memory and the "*then*" part specifies the actions to execute if the pattern is successfully matched. The RETE algorithm ensures that only those rules that are affected by changes in the working memory need to be reevaluated, which optimizes the system's performance and responsiveness.

One of CLIPS's strengths is its integration capability with mainstream programming languages, like *C* and *Java*. This feature not only allows CLIPS to operate within a host application, performing tasks and then returning control to the application, but also facilitates the embedding of CLIPS in larger software systems where complex decision-making processes are required. This enhances its utility in developing applications that require a mix of procedural and rule-based logic.

Since its origins, the use of CLIPS has extended beyond NASA and aerospace applications to sectors such as education, healthcare, finance, and manufacturing. In the academic context, it is employed to teach artificial intelligence and expert system concepts, while in the industrial one, it is used to develop sophisticated simulation models and decision support systems. The tool's ability to model complex decision-making processes makes it essential for creating dynamic and knowledge-based applications.

CLIPS excels as a significant development in the field of artificial intelligence tools, providing the required assistance in creating expert systems. Its comprehensive environment allows developers to craft detailed applications suited to a wide range of uses. The continuous evolution of CLIPS ensures it remains relevant in addressing the decision-making needs of modern systems, marking it as a central tool in both academic research and industrial applications.

2.2.1 CLIPS-Python binding

As specified in the thesis work of Bencivenga [6], **Clipspy** tool has been used as a "pythonic" linking layer between *CLIPS* and *Python*, to make native C APIs usable in Python environment.

The usage of `clipspy`, a *Python CFFI* (*C Foreign Function Interface*), brings significant advantages, like simpler syntax and error debugging, easier data structure implementations, and wide library support.

2.3 YAML

YAML (**YAML Ain't Markup Language**) is designed to be a human-readable data serialization format, widely utilized for configuration files, facilitating data sharing and enabling interprocess communication. Developed with the goal of simplifying the serialization process and enhancing readability, YAML is compatible with many agile programming languages including *Perl*, *Python*, *PHP*, *Ruby* and *JavaScript*, thanks to its *Unicode*-based design that ensures compatibility across diverse computing environments [7].

The structure of YAML is designed to reflect the conventional data structures found in programming, such as dictionaries (or hashes), arrays (or lists), and scalars (strings or numbers). This intuitive mapping, combined with YAML's syntax, renders it suitable for managing configuration files and debugging complex data structures. The language's structure is defined, indeed, by these three primitives:

- *Mappings*: these are used to create associations between unique keys and their corresponding values;
- *Sequences*: these represent ordered collections of elements, which may include a mix of strings, numbers, and other nested sequences or mappings;
- *Scalars*: these refer to individual data items, such as numbers or strings.

YAML employs *indentation* to define the structure of data, avoiding the brackets or braces common to many other data serialization formats like *XML*. This not only enhances its readability but also simplifies its use. Furthermore, YAML supports a versatile tagging system to maintain consistent data types across different programming platforms.

YAML is capable of single-pass processing, making it fit for streaming applications that demand quick and orderly data handling. It is provided to manage not only simple data structures but also more complex types. YAML can serialize and deserialize intricate data constructs, supporting features like object references and the management of cyclic references within object graphs.

The development of YAML was influenced by the need to overcome the verbosity typical of other data serialization formats. By focusing on efficient and user-friendly data representation, YAML addresses the needs of modern software development. It is especially valuable in scenarios that require strong configuration management and versatile data interchange capabilities, underlining its role in contemporary software applications.

2.4 TOSCA

Topology and Orchestration Specification for Cloud Applications (**TOSCA**) is an *OASIS* standard that redesigns how cloud applications and services are described, deployed, and handled. It mainly addresses the challenges associated with the deployment, management, and portability of cloud applications across various cloud platforms, aiming to cancel vendor constraints. This standard provides a comprehensive layout that details the relationships, dependencies, and operational behaviors of cloud services, thus enhancing the efficiency and scalability of cloud application management [8].

The initiative to develop TOSCA derived from the need for a robust framework capable of supporting both the deployment and consistent operational management of cloud applications over diverse cloud environments. This necessity was driven by the increasing complexity of cloud services and the market's demand for greater interoperability and flexibility in managing cloud resources. TOSCA's design aims to mitigate risks related to vendor limitations by allowing for a complete and standardized description of application and service topologies.

The primary goal of TOSCA is to simplify the lifecycle management of cloud services by automating deployment and operational tasks. It guarantees the modeling of applications across different cloud platforms using standardized specifications. The utility of TOSCA spreads across various cloud service models, including *IaaS*, *PaaS*, and *SaaS*, providing extensive orchestration capabilities that boost efficiency and agility.

2.4.1 TOSCA Simple Profile in YAML

Introduced to enhance accessibility and usability compared to the *XML* original format, the **TOSCA Simple Profile in YAML** employs the *YAML* language to simplify the expression of TOSCA templates. This profile offers a less complex approach to describe all aspects of cloud applications, from the underlying networks and servers to the relationships and operational policies governing their interactions [8].

Here are the basic components of TOSCA, essential for detailing the structure and management of cloud applications:

- *Service Templates*: the higher level of abstraction in TOSCA that encapsulates all definitions needed to orchestrate and manage an application, integrating various components;
- *Topology Templates*: detail the layout and relationships of nodes within the environment, defining how components are interconnected with each other;
- *Node Types*: these elements define the roles and capabilities of components such as compute instances, databases, or application modules, each provided with distinct functionalities and properties;
- *Relationship Types*: these specify the dependencies and interactions among nodes, which are essential for orchestrating the deployment sequence and managing operational dynamics;
- *Capabilities* and *Requirements*: this aspect of TOSCA details what functions a node can perform and the dependencies that must be met by other components or services, ensuring that each part of the system integrates properly;

- *Management Policies*: governed by rules or scripts, these policies determine the application's behavior throughout its lifecycle, addressing scenarios to maintain system integrity and performance.

An example of their use is shown in Figure 2.4.

TOSCA and its Simple Profile in YAML enable developers and administrators to deploy complex applications across multiple cloud environments. By establishing clear and reusable models, TOSCA minimizes errors, accelerates deployment times, and allows for automation and control over resources. Furthermore, TOSCA supports dynamic configuration changes and real-time policy updates, thus sustaining the resilience of cloud services.

TOSCA emerges as a useful standard in the cloud computing landscape, offering a structured and adaptable framework for the detailed description and automation of services and topologies, while the introduction of the Simple Profile in YAML has expanded access to advanced cloud orchestration capabilities.

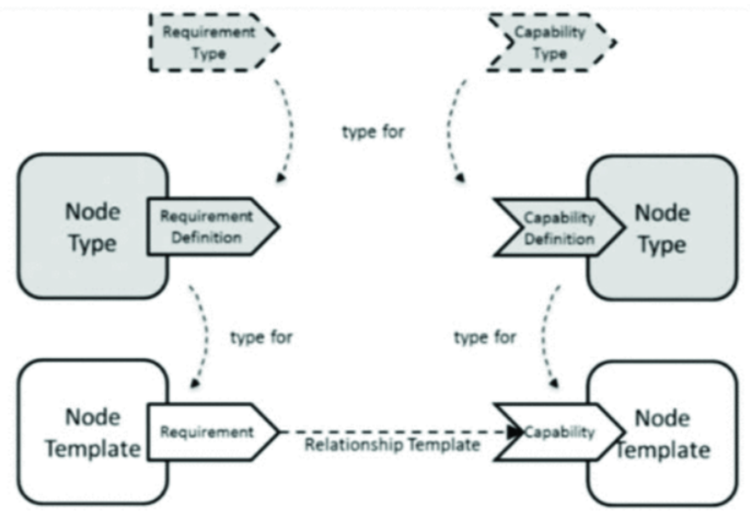


Figure 2.4: Examples of TOSCA Node Types and Templates [9]

2.4.2 TOSCA Node Types customization

In TOSCA, **Node Types** are necessary for defining the topology of cloud applications, serving as the building blocks that improve precise and effective cloud service orchestration. These Node Types are intrinsically designed to be adaptable, supporting the requirements specific to different environments. This adaptability is essential for integrating and managing diverse services and applications, which must align with particular operational standards and business needs [8].

TOSCA's architecture allows the customization of Node Types, making them suitable to meet the unique characteristics of each deployment scenario. By extending the base types provided in the TOSCA specification, developers can adapt the standard to diverse cloud environments and specific deployment needs, consequently maintaining compliance with the TOSCA framework.

The ability to derive Node Types is useful for integrating precise features or specialized behaviors not covered by the standard Node Types. For example, developers might add custom attributes or operations to a Node Type representing a certain network device or application component. This ensures that the model reflects the capabilities and requirements of the specific infrastructure.

2.4.3 TOSCA and Kubernetes integration

Introduction to Kubernetes

Kubernetes (**K8s**), developed by *Google* and now governed by the *Cloud Native Computing Foundation* (*CNCF*), plays as the major container-orchestration system designed to automate deploying, scaling and operating containerized applications. It supports multiple cloud environments such as *Google Cloud*, *Azure*, and *Amazon Web Services*. Kubernetes organizes containers into *pods*, manages their lifecycle, and automates their placement, scaling and monitoring across *clusters* of host machines [9].

TOSCA integration with Kubernetes

TOSCA defines the components of cloud applications and their relationships in a structured format, which **Kubernetes** does not natively support. By describing detailed application and service topologies, TOSCA helps in creating more organized and manageable cloud environments that can automatically handle application deployments and operational management [9].

Integrating TOSCA with Kubernetes typically involves translating TOSCA templates into Kubernetes object specifications, which requires an understanding of both frameworks' capabilities. The conversion process focuses on mapping TOSCA's structured syntax to the Kubernetes architecture, ensuring that all specified service and deployment requirements are represented in Kubernetes. This includes the translation of TOSCA's *capabilities* and *requirements* into Kubernetes' native constructs such as *services* and *deployments*.

The methodological approach for translating TOSCA into Kubernetes often involves defining rules that guide how each element of the TOSCA template is converted into corresponding Kubernetes objects. These rules might include:

- translating TOSCA nodes to Kubernetes pods or deployments, ensuring that runtime properties such as environment variables and container configurations are maintained;
- converting TOSCA relationships into Kubernetes services to establish correct networking between pods based on TOSCA's requirements and capabilities;

- implementing TOSCA's operational aspects within Kubernetes using policy configurations.

The integration of TOSCA with Kubernetes represents a strategic enhancement to cloud application management, combining detailed orchestration specifications with powerful container management. By leveraging TOSCA for application descriptions and Kubernetes for application deployment and operation, organizations can achieve more scalable and efficient application management.

2.5 JSON

JavaScript Object Notation (**JSON**) is a data interchange format inspired by *JavaScript* but language-independent, which has made it an ideal medium for information exchange on the web. JSON is structured as a collection of key-value pairs, where each key is a string and the value can be a string, number, boolean, array, or another JSON object, enabling hierarchical data structuring [10].

Due to its simple structure, JSON has become widely used for web APIs and configuration files, surpassing *XML* in popularity for many web applications. JSON's practical application fields extend from client-server data transmission to configuration setups across various software products and platforms.

This standard utilizes a schema-less nature, which simplifies the integration process but also requires explicit manipulation of data types and structures when exchanged across different systems. Despite its simplicity, JSON's utility in handling complex data structures and supporting nested data models provides significant flexibility and has been essential in developing *NoSQL* databases, which often use JSON for storing unstructured data.

In terms of querying, while there is no single standard query language for JSON, the development of several JSON query languages has enabled more sophisticated querying and transformations of JSON data, leveraging JSON's inherent structure to properly specify and retrieve information.

JSON's impact on data interchange and system configurations underlines its key role in current web technology, thanks to its simplicity and effectiveness in structuring and transferring data across diverse systems.

2.6 NSFs and Security Capabilities

Network Security Functions (**NSFs**) are basic components deployed to enforce and manage security policies within computer networks, especially in complex systems such as software-defined networks. These functions perform specific security tasks, e.g. blocking unwanted packet traffic or encrypting data flows, based on predefined rules that activate under specific conditions [11].

The concept of **Security Capabilities** refers to the features and functionalities that NSFs provide, enabling precise policy enforcement aligned to the specific needs of a network.

This flexibility of use involves a large range of security tasks and network environments.

A structured *model* of these security capabilities assists in the policy refinement process, offering a framework for defining and configuring the NSF operations. This includes translating high-level security policies into device-specific configurations.

The development of NSFs has evolved since their introduction in the early 2000s, driven by the complexities of network infrastructures and cyber threats. This evolution has been supported by organizations like the *Internet Engineering Task Force* (*IETF*) and its *Interface to Network Security Functions* (*I2NSF*) working group, which has been fundamental in standardizing frameworks and protocols for NSFs to ensure interoperability and security consistency across different systems and vendors.

In modern network security architectures, NSFs and their capabilities enable sophisticated management and automation of security protocols for protecting network resources. Their modular approach clarifies security management practices and also enhances the adaptability of security systems to emerging threats.

An example of NSF in practical application includes *iptables*, a widely recognized packet filtering system. The formal model describes iptables as capable of defining rules that establish packet forwarding or rejection based on IP addresses or protocol types, demonstrating how NSFs can be aligned to specific network scenarios.

The integration of standard models and continuous innovation provided by standardization bodies like the IETF, along with collaborative projects within the cybersecurity community, will ensure the effective addressing of evolving cyber threats and technological advancements. The next developments aim to move towards more comprehensive applications of NSFs, expanding their capabilities to incorporate advanced security measures such as VPN terminators, layer-7 filtering, and Web Application Firewalls.

2.6.1 NSF-Catalogue

Developed during Cirella's thesis work, the **NSF-Catalogue** sets up a *BaseX REST API Web Server*, which incorporates a *BaseX-based XML database* architecture. The catalog provides programmable access through tools like `curl` or the Python `requests` library. This server design focuses on flexibility and integration within automated workflows [12].

The primary objective of the NSF-Catalogue is to provide a unified interface that simplifies the management and configuration of NSFs from various vendors, by standardizing interactions within a common framework. This provides the automation of security policies across the network's structure, ensuring that security protocols are consistently maintained and reducing the need for continuous manual intervention. Additionally, the catalogue is designed to assist users in configuring NSFs despite vendor-specific implementations, minimizing potential errors and enhancing the reliability of security setups. By offering a single, cohesive interface, the NSF-Catalogue brings uniformity to interactions with NSFs across complex software networks.

The NSF-Catalogue allows for verification of which NSFs are available within the network, assessing the compatibility of different NSFs based on shared or unique capabilities and determining which NSFs can implement specific rule instances. Such functionality enhances strategic planning in security management and provides detailed insights into each NSF's capabilities, assisting administrators in making decisions when selecting NSFs to meet specific security demands. This ensures an optimized deployment of resources suited to the network's security needs. For these purposes, the NSF-Catalogue includes a set of *APIs* designed for efficient NSF querying:

- *Comparison of Two NSFs*: gets the names of two NSFs and examines the relationship between their Security Capability sets, assessing whether one is fully contained within the other, if they are equivalent, or if they have no overlap.
- *NSF Search Based on Rules*: accepts the file path of a specific *RuleInstance* and identifies the NSFs capable of implementing the policies within it.
- *NSF Search by Security Capabilities*: by taking a specified set of Security Capabilities as input, this API returns a list of NSFs that possess these capabilities.
- *List All Security Capabilities*: outputs a full list of available Security Capabilities within the catalogue, organized by type. This feature offers a comprehensive overview of the available options.

2.7 Knowledge Base management

In complex systems, **knowledge bases** are vital for collecting, analysing, and reusing information to optimize processes and reduce repetitive work. By establishing a logical framework for managing large sets of data, knowledge bases make possible the organized retrieval and application of information in decision-making, enabling to leverage past insights within ongoing workflows. This approach supports continuous learning from previous tasks, helping to avoid redundant actions and optimize efficiency [13].

A well-designed knowledge base enhances data handling by making essential information available at different stages of a process. Through a proper structure, knowledge bases reduce the need to repeatedly gather information or revisit decisions. This structure not only raises consistency across subsequent phases but also boosts workflow performance by enabling systems and users to rely on stored knowledge.

Such an approach enhances resource management by serving as a centralized source of truth that evolves with new insights. This flexibility allows processes to respond efficiently to changes, reducing redundancy and improving decision-making.

Chapter 3

Related Works

This chapter explores previous studies depicting the basis on which this thesis builds upon. It begins with a description of the framework presented in Bencivenga's thesis for the automatic refinement of HSPLs into configurations executable across network devices, featuring the Refinement, Converter, and Orchestrator tools to handle each phase. Additionally, the Controller component in the FISHY project plays a crucial role in translating high-level security intents into medium-level configurations adapted to specific NSFs. Finally, a focus on how the Security Capability Model (SCM) supports the policy refinement process, enhancing automation and minimizing errors across diverse network environments.

3.1 HSPLs Automatic Refinement Framework

This thesis reconnects to the foundational work of Bencivenga, who developed a structured framework for the automatic refinement of *HSPLs* across network environments. Bencivenga's thesis represents a starting point, aiming to automate the translation of abstract HSPLs into configurations executable by *NSFs* and enforceable by network devices. The framework is built on three main components: *Refinement tool*, *Converter tool*, and *Orchestrator*. Each of them deals with different stages in the policy translation process. Together, they contribute to a cohesive system capable of transforming high-level security directives into actionable device configurations [6].

A schema of the framework workflow is provided in Figure 3.1.

3.1.1 Refinement tool

The **Refinement tool** is central to Bencivenga's framework, handling the primary task of converting high-level security policies into *intermediate* configurations. This tool leverages *Clipspy*, a Python interface for the *CLIPS* inference engine, to apply rule-based logic in parsing and transforming HSPLs into configurations that align with the specific security capabilities of network devices. During this translation process, the Refinement tool

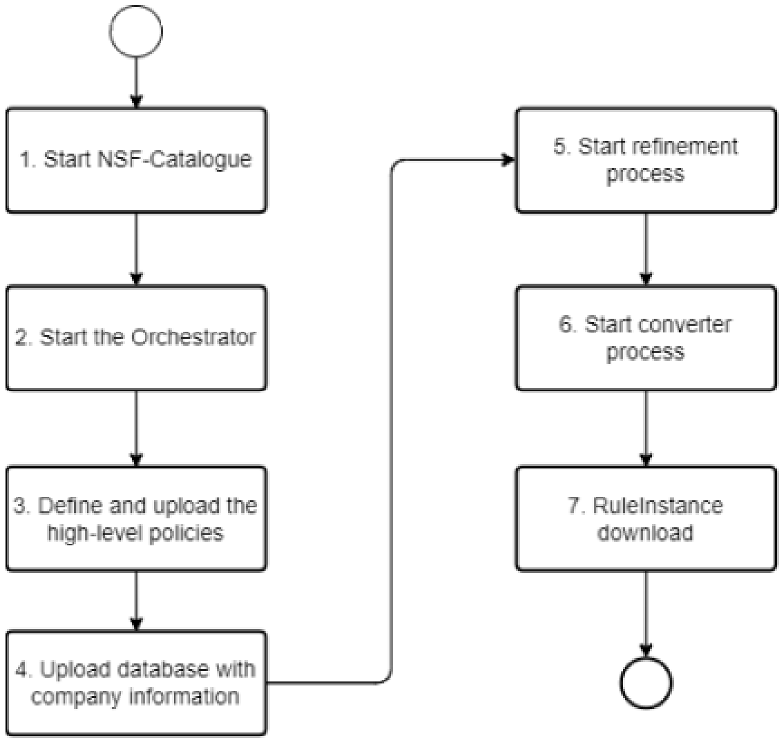

1. Start NSF-Catalogue
2. Start the Orchestrator
3. Define and upload the high-level policies
4. Upload database with company information
5. Start refinement process
6. Start converter process
7. RuleInstance download

Figure 3.1: Refinement Framework Workflow

examines the provided HSPLs to identify necessary security controls and then matches these controls with the capabilities of available NSFs.

Key steps involved in the Refinement tool's operation include:

- *Requirements identification*: the tool initially processes the HSPLs to identify the specific security capabilities required for their enforcement. This step involves exploring the policy to determine which security controls are needed and aligning these with the capabilities available in the network;
- *Non-enforceability analysis*: in cases where an HSPL cannot be implemented due to a lack of corresponding capabilities or network constraints, the tool conducts an analysis to identify these limitations, ensuring that policy enforcement is feasible with existing resources;
- *Device capability matching*: once the necessary capabilities are confirmed, the tool performs a detailed matching of these requirements with the available NSFs within the network, verifying that the chosen devices meet the policy's technical needs;
- *Device selection*: if multiple NSFs possess the required capabilities, the Refinement tool evaluates them and selects the optimal device(s) for policy implementation. This selection is based on compatibility, performance considerations, and any predefined rules in the system's configuration;

- *Intermediate file generation*: the tool generates an organized JSON file containing the intermediate configurations. This file provides a representation of the policy, which subsequent tools can further refine and convert into device-specific configurations.

Implementation overview

The main implementation components and files within the Bencivenga's Refinement tool include:

- *Company database*: this database, defined in `company_database.py`, contains information specific to the organization's network infrastructure. Key entries include:
  - *Network backbone and topology*: defines the links among subnets and devices, providing a clear structure of the network's connectivity;
  - *Device capabilities*: lists NSFs installed on each device, enabling the tool to match policies to specific security functions available in the network;
  - *Protection algorithms*: specifies chosen algorithms for encryption and authentication with their configuration sets;
  - *Info database*: stored in `info_database.py`, this secondary database complements the company-specific database, including mappings for high-level policy objects, such as network traffic types (e.g. VoIP or DNS traffic) and their associated protocols or ports;
- *CLIPS rules*: located in `rules.py`, this file comprises the rules leading the CLIPS engine. The rules are organized based on common policy actions (e.g. filtering, confidentiality protection) and include forward-chaining logic that triggers when specific conditions are met. Each rule can assert new facts or call Python functions with the correct parameters;
- *Network graph generation*: the tool leverages the `NetworkX` and`Pyvis` libraries to generate an interactive graph of the network. NetworkX provides structural insights on the network, helping to locate subjects and objects in relation to each other, while Pyvis creates a visual interface that can display or modify the topology if enabled;
- *Device configuration selection*: once the tool identifies the required capabilities for a given policy, it queries the *NSF-Catalogue*, retrieving NSFs that match the policy requirements. Then, it checks device compatibility along the network path;
- *Output generation*: the final output of the refinement process is a JSON file, organized for the next stage in the workflow thanks to specific functions managing the output format. *Filtering* rules generate two configurations, while *protection* rules produce four.

3.1.2 Converter tool

The **Converter tool** takes the intermediate file produced by the *Refinement tool* and further translates it into suited configurations for the devices, ready for implementation by NSFs. This tool systematically processes each segment of the provided input file, translating the policy components into *RuleInstance* XML files appropriate to each device's capabilities.

In its initial phase, the Converter tool parses the intermediate JSON file, interpreting each NSF's designated section. During this parsing, the tool examines the capabilities required by each policy block, converting them into device-level configurations that align with the capabilities of the identified NSFs. This process ensures a direct mapping from high-level security intentions to low-level configurations compatible with the device infrastructure.

Another essential aspect is the management of redundancy. When multiple NSFs are available for a specific policy requirement, the tool selects a single NSF per task to optimize resource usage. It achieves this by appending configurations to an existing RuleInstance file for an NSF if it is already assigned similar tasks, therefore reducing duplication and maintaining coherent deployment across the network. This aggregation process helps in aligning the translated configurations with the network's operational needs, avoiding overlap and redundancy in rule applications.

These mechanisms ensure efficient and consistent policy enforcement by directly adapting policies into detailed and device-specific commands, avoiding unnecessary duplication of configurations and enabling scalable deployment across the network.

3.1.3 Orchestrator

The **Orchestrator** serves as the centralized management component of the framework, supervising the execution of the *Refinement* and *Converter tools*. Implemented as a *Flask*-based server, the Orchestrator provides a *REST API* interface, enabling users to interact with and control the policy refinement process through a set of well-defined endpoints.

The main functions provided by the Orchestrator include:

- *Input files upload*: allows administrators to upload policy files and network topology information necessary for policy processing;

- *Execution management*: offers options to initiate the Refinement tool with or without a Graphical User Interface (GUI);

- *Output files download*: provides endpoints for downloading the finalized RuleInstance files, either individually or as a set.

Through these functionalities, the Orchestrator integrates the framework's components, enabling administrators to efficiently manage the policy refinement lifecycle from high-level abstraction to device-level configuration.

3.2 FISHY Project

The **FISHY project**, funded by the *European Union's Horizon 2020 program*, aims to create a secure framework for managing and safeguarding complex ICT-based supply chain systems. By integrating security assurance, certification, and dynamic policy enforcement tools, FISHY provides a comprehensive approach to protect network components. In collaboration with various partners, among which the *Politecnico di Torino*, the project emphasizes adaptive security measures to address emerging cyber threats in real-time and supports compliance with security standards and legal obligations [14].

The framework includes two main modules:

- *Security Assurance and Certification Management* (*SACM*), which verifies and certifies security across the infrastructure.
- *Enforcement and Dynamic Configuration* (*EDC*), which applies and adjusts security policies to fit specific network devices and configurations.

Together, these modules enable FISHY to provide continuous security monitoring and policy adaptation, maintaining network integrity and resilience.

3.2.1 Controller

The **Controller** within the *EDC* module is a crucial component for transforming high-level security policies into device-enforceable configurations, specifically suited for deployment across NSFs. Acting as the central refinement engine, the Controller translates abstract policies by interpreting each policy's security requirements and then producing medium-level configurations that align with the specific capabilities of target NSFs.

Key operations of the Controller include:

- *Policy analysis and requirement identification*: it evaluates each high-level policy, decomposing its components to determine necessary security controls;
- *NSF selection and policy allocation*: based on defined capabilities, it identifies compatible NSFs within the network to enforce specific policy requirements, optimizing the selection to meet network security needs;
- *Intermediate policy generation*: the Controller then produces medium-level policies, outputted in an intermediary format. These policies maintain essential security requirements but are adapted for further processing and enforcement by the network's NSFs.

The Controller not only adapts policies to NSF capabilities, but also enhances the framework's flexibility by allowing dynamic adjustments to security configurations in response to network changes or new security requirements. This component's role is essential to the FISHY framework's ability to align broad security goals with specific and

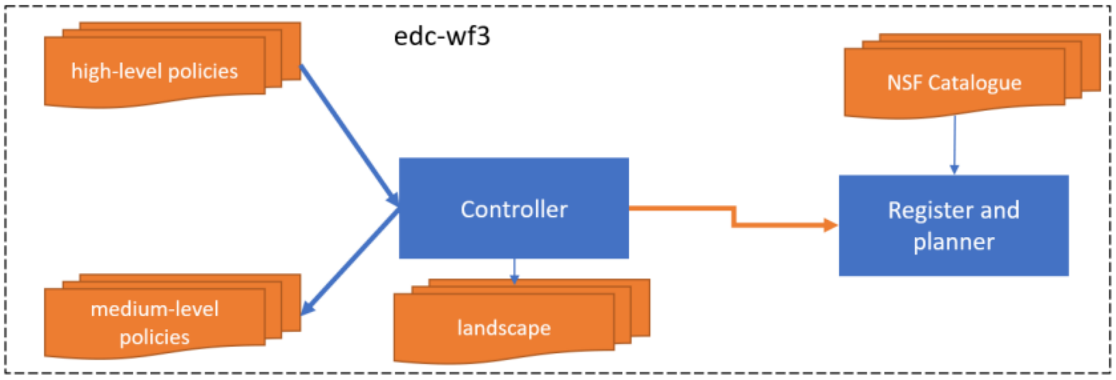


Figure 3.2: Controller Workflow

enforceable policies across diverse network environments, enabling scalable and proactive network security management.

For visual representation, see Figure 3.2.

Controller's APIs

The *Controller* operates as a web service, built using the *Flask* framework, and is accessible through both a GUI and several API endpoints via *CURL* syntax. These APIs support essential tasks for refining and deploying security policies across network environments:

- *Policy file upload*: allows administrators to upload HSPLs in XML format for processing and refinement;
- *Network landscape upload*: enables the upload of network layout information, essential for determining network enforcement points;
- *Interactive refinement execution*: starts an interactive session where administrators select appropriate NSFs for policy enforcement, with the Controller refining details based on chosen options;
- *Automated refinement execution*: initiates an automatic NSF selection process, using default criteria to apply policies;
- *Policy conversion*: transforms refined policies into medium-level configurations, outputting a list of configured NSFs;
- *RuleInstance file download*: allows downloading device-specific configuration files for NSFs, supporting both single and batch downloads.

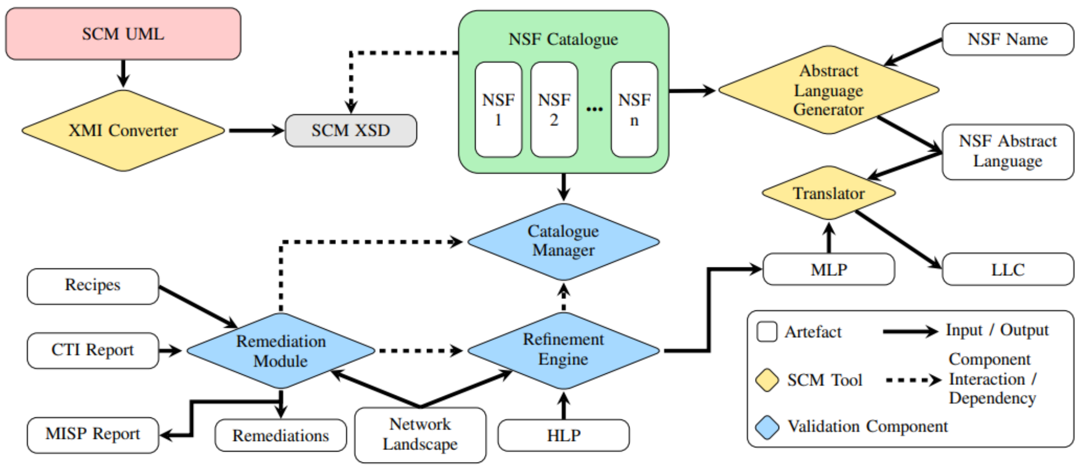


Figure 3.3: Overview of interactions between components built upon the SCM

3.3 Security Capability Model

The article proposed by Basile, Settanni, and Gatti, introduces the **Security Capability Model** (**SCM**), a formal framework designed to standardize and simplify the configuration of **Security Controls** (**SCs**) like firewalls and VPNs, especially in complex and multi-vendor network environments. SCM addresses the frequent configuration errors and challenges posed by manual error-prone processes. By abstracting and modeling SC functionalities, SCM provides a structured approach that translates HSPLs into specific configurations for devices, enabling automation and reducing dependency on particular vendors. SCM's dual-layer structure, based on an *Information Model* and a *Data Model*, ensures both a vendor-neutral representation and a consistent integration of various SC types, making it suitable for policy refinement and incident response applications [15].

A graphical overview is displayed in Figure 3.3.

3.3.1 Impact on Policy Refinement

One of SCM's key contributions consists of its support for *policy refinement* about translating HSPLs into actionable configurations for diverse NSFs. SCM achieves this by using a model-driven approach that maps abstract security requirements directly to device-specific commands, addressing the unique capabilities of each SC.

The SCM's structure assists administrators in selecting and configuring devices based on the precise alignment of SC capabilities with specified security needs, simplifying the deployment of policies across a heterogeneous network landscape. This process not only minimizes manual interventions but also reduces configuration errors and conflicts, which are common in traditional SC setups. Moreover, the model-driven approach improves the selection of suitable security controls and the accurate translation of policies into enforceable configurations, demonstrating its value in the policy refinement process across network security devices.

Chapter 4

Problem Statement

4.1 Introduction

The starting point of this thesis is the *automated HSPL refinement tool* mentioned above, an evolving framework that has seen continuous development through various code integrations and contributions from different interested parties. Despite these incremental advancements, the tool still exposes some limitations, particularly concerning optimization and flexibility aspects, but also areas regarding robustness and security itself. These disadvantages become increasingly significant as network environments grow in complexity, and the demands on security policy refinement tools get more rigorous.

As networked systems scale, achieving a balance between adaptability and precision in security policy implementation is crucial. While functional, the existing tool requires further enhancements to meet these new demands, both possibly boosting performance and without introducing vulnerabilities. This research, therefore, is dedicated to identifying and addressing specific gaps in the current framework.

This problem statement thus sets the stage for a focused investigation into how specific advancements can be integrated into the HSPL refinement tool to expand its use cases. The ultimate objective is to evolve the tool into a more versatile and resilient asset for automated security policy refinement, better suited to address the dynamic needs of modern networked environments.

4.2 Scenarios

To provide a solid basis for these improvements, this thesis establishes a series of scenarios that were chosen to represent common challenges in policy refinement across different network environments. These scenarios were selected to reflect real-world limitations and complexities, serving as reference points for evaluating the effectiveness of the proposed solutions. Each scenario addresses specific aspects of the refinement process, identifying opportunities for enhancing flexibility, scalability, and security within the framework.

Building on these scenarios, the work aims to deliver solutions that are not only theoretical, but also practical and adaptable to diverse deployment contexts, ensuring their relevance in facing current and emerging needs.

The discussed scenarios will address the following topics:

1. introducing a standard language to formalize the network layout description;
   - (a) handling typed network components' data structures;
   - (b) providing a cloud-native format;
2. splitting security conditions across multiple security controls;
3. supporting stateful policy management;
   - (a) providing endpoint protection.

Each scenario will contribute to the overall goal of creating a better refinement framework by addressing specific challenges through the proposed advancements.

4.2.1 Standard Model for Network Layout

The first scenario focuses on the need for a standardized and robust model to describe the network layout, including its components, the connections between them, and all the specific properties and functionalities of each network element. The network information was previously contained within a Python file named `company_database.py`, serving as a data repository. This approach presents multiple issues.

Embedding network configurations in a Python script lacks formalized structure, leaving room for misinterpretation and inconsistent use, especially in cases where multiple developers or administrators need to update or review the network layout. Each entry might differ in format or content without an established schema, making it hard to consistently parse or verify the network data. Furthermore, without validation mechanisms, such a format offers no inherent checks to test the accuracy or completeness of the information being used.

This approach raises security concerns, too. Storing network layout details within an executable Python script makes the tool vulnerable to potential code injection attacks. If unauthorized modifications are introduced into the Python file, they could alter the tool's behavior, potentially leading to critical security breaches.

Given these limitations, it is clear that it is necessary to introduce a standardized model to represent network configurations in a more rigorous way, described with a formalized format to minimize risks of inconsistency and vulnerability. A validation mechanism will also be implemented to ensure the accuracy and integrity of the model, providing checks for completeness and compliance of network data across various cases.

Typed data structures

Once a formal model is defined, it becomes essential to categorize the network components according to their types, establishing clear distinctions that reflect the specific attributes and roles of the elements within the network. By defining these classifications, we can derive type-specific data structures representing each network entity according to its functional category, ensuring a consistent framework for managing varied network parties.

These typed structures, such as Python dictionaries or classes, serve as templates for each category, specifying the expected attributes and functionalities corresponding to each component's role within the network. By associating relevant characteristics with respective types, this approach provides a structured way to define and access each element's unique attributes, supporting a consistent and organized representation of network entities.

Through these specific data structures, the tool gains the ability to apply processing and analysis logic suited to different categories, ensuring that each element is managed in a way that aligns with its intended purpose. This classification not only enhances the accuracy and efficiency of component handling, but also provides flexibility and scalability in managing diverse network architectures and security functions.

Cloud-native and K8s integration

In addition to addressing the limitations of the approach shown above, the new format for describing the network layout should be designed to align with *cloud-native* principles and *Kubernetes* conventions. As *K8s* has become a de facto standard for managing containerized workloads, ensuring compliance with its paradigms is essential for increasing the scalability and interoperability of the tool in modern deployment environments.

Introducing such a format enables the framework to integrate with Kubernetes native resources. This alignment allows the tool to adapt to evolving network configurations, ensuring it can handle dynamic changes in modern infrastructures while maintaining consistency with Kubernetes-based deployments.

By embedding support for K8s, the tool will meet industry standards and improve its ability to operate in diverse deployment scenarios. This approach ensures the framework remains relevant in cloud-native environments, making it a reliable solution for managing scalable and distributed architectures.

4.2.2 Complex Security Filters

The second scenario concentrates on addressing the limitations of the current framework when handling complex security conditions that cannot be enforced at a single security control. The framework assumed that all security requirements specified in a policy could only be implemented by a single NSF. However, this assumption reduces the flexibility of the tool in scenarios where certain security conditions exceed the capabilities of an individual NSF or when constraints in the network infrastructure prevent such enforcement.

The introduction of "*complex filters*" is proposed to overcome these challenges. This concept involves distributing the enforcement of security conditions across multiple security controls, allowing the system to handle policies that cannot be fully enforced at a single point. By splitting the required conditions among different NSFs, the framework can ensure that security policies are implemented even in constrained environments.

This distributed enforcement mechanism aims to enhance the tool's ability to differentiate between three distinct cases:

- *Full enforceability*: when the policy can be deployed entirely by splitting the conditions among different security controls;
- *Enforceability with required monitoring*: when the policy conditions are split anyway, with monitoring to estimate residual risk;
- *Non-enforceability*: when the policy cannot be implemented even through distribution, and appropriate alerts or fallback mechanisms are required.

By enabling the splitting and distribution of security conditions, the framework achieves greater versatility and adaptability. This improvement ensures that security policies remain actionable in scenarios where the traditional approach would fail due to technical or infrastructural constraints, thus expanding the range of policies that can be supported by the tool.

Implementing this mechanism will both improve the framework's ability to handle diverse security scenarios and provide network administrators with higher control over the enforcement process. The ability to split security conditions permits the tool to align with the complexities of modern network environments, where a single point of enforcement may no longer be sufficient.

4.2.3 Stateful vs Stateless Framework

The third scenario relates to upgrading the framework's capabilities by introducing *statefulness*. This improvement would allow the tool to distinguish between stateful and stateless policies, leveraging the benefits of stateful processing to achieve optimal outcomes in refining and enforcing security policies.

While stateless processing offers simplicity and scalability, it lacks the contextual awareness needed for fine security decisions. Vice versa, stateful processing provides a richer context for decision-making, but requires careful management of state data to avoid performance bottlenecks or inconsistencies. Balancing these trade-offs is the key for implementing a robust framework.

Stateful capabilities would enable the framework to evaluate policies against the past network state, adapting its decisions dynamically. By maintaining knowledge of ongoing conditions and previous interactions, the tool could make more informed choices, aligning the refinement processes with the system's real-time needs. This adaptability guarantees that the framework can respond to changing circumstances, providing a more robust and versatile approach to policy management.

Furthermore, introducing statefulness leads to adaptive policies that evolve based on the current state of the network. These policies would both enhance security by addressing dynamic threats and improve performance by optimizing resource allocation and minimizing unnecessary configurations. By embedding stateful logic, the framework could conciliate resiliency and optimization, conforming security management with the dynamic nature of modern network environments.

This scenario represents a significant step towards making the framework more intelligent and responsive, assuring that it can meet the requirements of continuously changing deployment scenarios.

Endpoint security controls

Configuring security controls individually for endpoints is often too resource-intensive and impractical, particularly in large networks. Endpoints, often being the most numerous and varied elements in the network, require a more simplified approach to ensure effective protection without excessive complexity.

Some proposed solution involves adopting best practices to secure endpoints, such as:

- *Zero Trust*: requiring verification for all actions and assuming no endpoint is intrinsically secure;
- *Default Deny Policies*: enforcing a deny-all baseline, allowing only explicitly permitted actions or communications.

These strategies streamline endpoint management while maintaining strong security. By focusing on recommended practices rather than custom configurations, the framework would ensure robust protection that scales efficiently, even in evolving environments.

Chapter 5

Design Overview

The chapter describes the design of the solutions implemented to address the limitations identified in the *Problem Statement* (Chapter 4) and enhance the performance of the HSPL refinement tool. The work has focused on three main areas of improvement:

- adopting the *TOSCA YAML Standard* to formalize the network layout representation and enable validation and interoperability;
- extending the mechanism for the *NSF-Catalogue querying* to improve the matching process of security policies with the capabilities of available Network Security Functions;
- integrating a *knowledge base* to support adaptive reasoning for policy refinement, allowing dynamic context and state-aware decisions.

Additionally, various incremental improvements have been made throughout the development, addressing some operational and refactoring aspects. Each of these areas will be explored in detail in the following sections.

5.1 TOSCA YAML Standard for Network Layout

5.1.1 Motivation

The decision to adopt **TOSCA YAML** as the standard for describing the network layout was based on its ability to address the tool's specific requirements. After evaluating multiple alternatives, TOSCA YAML was selected for its strengths in representing network topologies and associated component information in a clear and structured manner:

- *Detailed modeling*: TOSCA provides a framework for accurately describing network components, their relationships, and configurations. This ensures the layout and associated information can be gathered comprehensively;

- *Interoperability*: as a vendor-neutral specification, TOSCA supports integration with diverse tools and platforms, an essential feature for environments involving multiple technologies and providers;
- *Extensibility*: the modular nature of TOSCA allows for customization to include specific network elements and relationships as needed, guaranteeing the tool can adapt to future requirements;
- *Validation*: the schema-based structure enables validation of the network description, both assuring consistency and reducing the risk of errors;
- *Cloud-native orientation*: by design, TOSCA is suitable for describing dynamic and cloud environments, making it compatible with modern deployment platforms, e.g. *Kubernetes*.

The use of *YAML* as the underlying format further boosts its applicability for describing the network layout:

- *Simplicity*: the human-readable syntax simplifies the creation and understanding of network descriptions, minimizing errors and reducing the effort required to manage configurations;
- *Compactness*: its lightweight structure avoids the verbosity of formats like XML, making it more suitable to represent, for example, complex hierarchies;
- *Tool compatibility*: YAML is widely supported by modern tools and frameworks, including *Python*, which ensures uniform integration with the HSPL refinement tool.

5.1.2 Information Structuring

The TOSCA model adopted for the description of the network layout has been structured into two distinct files, each serving a specific purpose.

Component type definitions

The first file is dedicated to defining the types of components that may exist within the network. This contains customized types derived from the TOSCA's base types, fitted to the tool's needs. These types specify the attributes and capabilities associated with different elements and the kinds of connections they support.

Key aspects of this file include:

- *Node types*: the types of nodes required for the network layout have been defined to represent the following entities and their functionalities:
  - *Device*: nodes like firewalls or VPN gateways that implement security functions;
  - *Subnet*: representing network subnets containing multiple connected elements;

  - *User (host)*: regular network users or endpoints;
  - *Malicious user*: specific nodes for modeling potential threats or controversial behaviors;
  - *Web application*: representing hosted applications and services;
  - *Traffic types*: defining different categories of network traffic, which can be used to simulate flows;

- *Connection types*: the model specifies the kinds of relationships and connections between nodes, representing various forms of network connectivity and dependency;
- *Capability types*: certain node types, like devices, are further equipped with capabilities reflecting the specific security functions (*NSFs*) they can support.

This file provides a reusable and modular foundation for constructing network layouts, by offering a template for all node and capability types.

Network component instances

The second file builds upon the types defined in the other one, describing the actual components and connections of the network topology being modeled. This file includes:

- *Node instances*: each node is an instance based on the types defined in the first file, with attributes populated to represent specific entities within the network;
- *Network topology*: connections between nodes are defined here, establishing the network topology. Each link specifies which nodes interact with one another and the nature of their relationships, aligning with the connection types defined in the first file;
- *Attribute values*: the nodes' attributes are specified to reflect the current network configuration.

This separation between the definition of types and the instantiation of network components allows for a clear and flexible design. The first file serves as a schema, while the second one provides the concrete details of the network being modeled, improving maintainability and also simplifying future updates and extensions to the network layout.

5.1.3 Validation Schemas

To ensure the integrity and correctness of the *TOSCA YAML files* used for describing the network layout, a validation mechanism based on *JSON schemas* has been implemented. These schemas play a crucial role in defining a structured and consistent format for the two main files: the *component type definitions* and the *network component instances*. By imposing clear rules and required fields, the schemas establish a quite rigid structure, enabling the TOSCA YAML files to conform to predefined structures.

For the type definitions file, the schema forces all types of components and their attributes to be properly defined, including capabilities and valid connection types. In the same way, for the instances file, it verifies that the network topology is described following the right pattern. This dual-schema approach also determines the mandatory sections that both files must include.

Key validation constraints

The validation process enforces several constraints during file creation, assuring compliance with predefined rules, like the required types for each field.

Furthermore, the following options are directly presented to the user as "*allowed values*" during the writing phase, making it easier to adhere to the required format:

- *Supported NSFs*: only NSFs defined in an external *XML file* are allowed as node supported security functions. These NSFs are dynamically derived from the *NSF-Catalogue*, an XML file preloaded from the repository of the *Security Capability Model* discussed in the *Related Works* (Chapter 3). This file is parsed to populate the schema with valid options, guaranteeing that the selection aligns with recognized standards;
- *Node types*: derived types must follow the hierarchy established by the TOSCA standard, maintaining compatibility and adherence to best practices;
- *Connection types*: relationships between nodes are restricted to predefined types, preserving the validity of the network model.

Real-time validation

One of the main advantages of using schemas is the ability to provide automatic validation and real-time feedback while creating the TOSCA YAML files. The schema acts as a control mechanism, immediately flagging issues when the structure or content does not meet the established constraints. For example, unsupported node types, forbidden attributes, or invalid connections are reported directly during editing. This automatic feedback ensures errors are caught early, reducing the need for subsequent corrections and speeding up the overall workflow.

By embedding this automated control into the writing process, the tool minimizes human error, maintains consistency across files, and secures adherence to the expected format. This level of integration enhances the reliability of the network descriptions and reduces the effort required for manual validation.

5.1.4 Files Parsing

The parsing of the *TOSCA YAML files* is managed separately for the *type definitions* and the *instance descriptions*, utilizing dedicated *Python scripts*. Each script focuses on its respective file, guaranteeing that the information is processed and organized into dynamically created data structures, enabling accurate representation and further processing of the network layout.

Parsing type definitions

The first parsing script processes the file containing component types' definitions and associated capabilities. Its primary objective is to extract and organize the properties of each node type, considering the TOSCA standard's hierarchical structure too.

Key steps include:

- *Collecting node properties and capabilities*: the script identifies and gathers all attributes and capabilities of different node types. This process also accounts for inheritance by tracing the hierarchy of each type back to its TOSCA base type, ensuring that standard attributes are incorporated along with custom ones;
- *Dynamic data structure types generation*: using the collected information, the script dynamically creates proper types for data structures that encapsulate the proper attributes and capabilities. These classes will be used in the other parsing script for the instantiation of network elements.

Parsing network layout

The second script processes the file that describes the instantiated components of the network. Leveraging the types generated by the other script, it derives data structure instances, populating their attributes with correct values, and organizes them into categories.

Key steps include:

- *Instantiating data structures*: the script produces data structures based on the classes defined in the first parsing script, populating them with collected attribute values;
- *Instance categorization*: all instances are sorted into three main categories, designed to align with the specific requirements of the tool:
  - *Subnets*: representing network segments with a defined netmask, to which other nodes can be connected;
  - *Devices*: nodes implementing specific security functions (e.g. firewalls or VPN gateways);
  - *Entities*: remaining node types, such as users, malicious users, and others;
- *Parsing connections and topology*: the script parses the connections between nodes and organizes them into a data structure representing the network backbone. This enables the creation of a connection graph, which can also be visualized for enhanced understanding and analysis.

Validation through parsing

An important benefit of using dedicated *libraries* for *YAML* and *TOSCA* file parsing is the additional layer of validation they provide. These libraries make sure that the TOSCA YAML files adhere to syntactic and structural rules, complementing the *JSON schemas* validation. Any inconsistencies, such as incorrect attribute names or unsupported structures, are signaled during parsing, ensuring that only valid configurations are processed.

This multi-validation approach improves the reliability of the network descriptions, minimizes potential errors, and reinforces the integrity of the entire workflow.

5.1.5 Refinement Process Alignment

The transition to the *TOSCA YAML-based network layout model* required updates to the *refinement tool's code* to handle the structured data produced by the parsing scripts. Functions responsible for analyzing, managing, and retrieving network information were modified to work with the new data structures.

Function updates allow for the processing of the organized elements' attributes and consistent interpretation of relationships between nodes. Furthermore, the categorization into subnets, devices, and entities, as well as the defined topology, enables a more efficient information extraction.

These adjustments were necessary to align the refinement tool with the model-based approach, permitting it to operate with the formalized and validated framework correctly. The revised code now supports the improved standard, providing scalability and accommodating potential extensions to the network layout or its attributes.

5.2 Extended Strategy for NSF-Catalogue Querying

To address the constraints related to enforcing complex security conditions described in the *Problem Statement* (Chapter 4), this design phase introduces a novel approach for **querying the NSF-Catalogue**. The primary goal is to extend the original mechanism, which searches for individual *Network Security Functions* capable of fulfilling all the requested conditions, to include the possibility of identifying *combinations of NSFs* that collectively satisfy the required set of capabilities.

This enhancement guarantees that even when no single NSF can independently support all the requested conditions, the tool can still provide valid solutions. By allowing the distribution of security conditions across multiple NSFs, the framework increases the chances of successful policy implementation in scenarios where individual NSFs face limitations in their supported capabilities.

The proposed approach focuses on the following objectives:

- *Main target*: prioritize finding one or more single NSFs that fulfil all requested conditions, permitting the simplest and most efficient solution whenever possible;

- *Fallback mechanism*: when no single NSF can satisfy all conditions, identify combinations of NSFs that together support the entire set of requested capabilities;
- *Output optimization*: generate a list of reasonable NSF combinations, ensuring the results are practical. For instance, combinations containing an excessive number of NSFs for a single policy are avoided as they may be inefficient or unrealistic to implement;
- *Capability mapping*: provide an overview of the capabilities satisfied by each NSF in the results, with a clear mapping in the form of `NSF: list of supported capabilities`. This mapping assures transparency and aids in understanding how each NSF contributes to the policy implementation.

This upgraded querying mechanism raises the refinement tool's flexibility by offering a larger range of actual possibilities. The output includes:

- a prioritized list of NSF combinations that satisfy all requested capabilities;
- detailed information for each NSF in the combinations, including the specific capabilities it supports.

The flowcharts in Figure 5.1 illustrate two different results from querying the NSF-Catalogue during the refinement process, focusing on a policy requiring specific entity identifiers. The IDs in this example include *IP address*, *MAC address*, *Distributed ID* (*DID*), and *Wallet ID* (*WID*). In the top section, the previous situation is depicted, where the *Refinement Engine* queries the *NSF-Catalogue* to find a single NSF capable of satisfying all the required conditions. As no NSF supports all the requested identifiers simultaneously, the process fails, resulting in a "*No NSF Found*" outcome. This demonstrates the rigidity of the prior approach, which cannot handle scenarios where a single NSF is insufficient to meet all the policy conditions.

Vice versa, the bottom section presents the enhanced mechanism designed to overcome this limitation. Instead of looking for a single NSF, the *Refinement Engine* evaluates combinations of NSFs that collectively satisfy the required conditions. In the example shown, the conditions are split between two NSFs: *IpTables*, which supports the capabilities concerning the *IP address* and *MAC address*, and *ethereumWebAppAuthz*, which addresses the *Distributed ID* and *Wallet ID* requirements. This allows the system to identify a valid solution by mapping specific capabilities to the appropriate NSFs, outputting a list of feasible combinations.

This approach extends the framework functionalities by enabling the distribution of conditions across multiple NSFs, making it adaptable to complex scenarios and providing wider applicability in real-world environments.

5.3 Knowledge Base Integration

To increase the adaptability and efficiency of the HSPL refinement tool, the integration of a **knowledge base** has been introduced. This addition allows the tool to maintain and

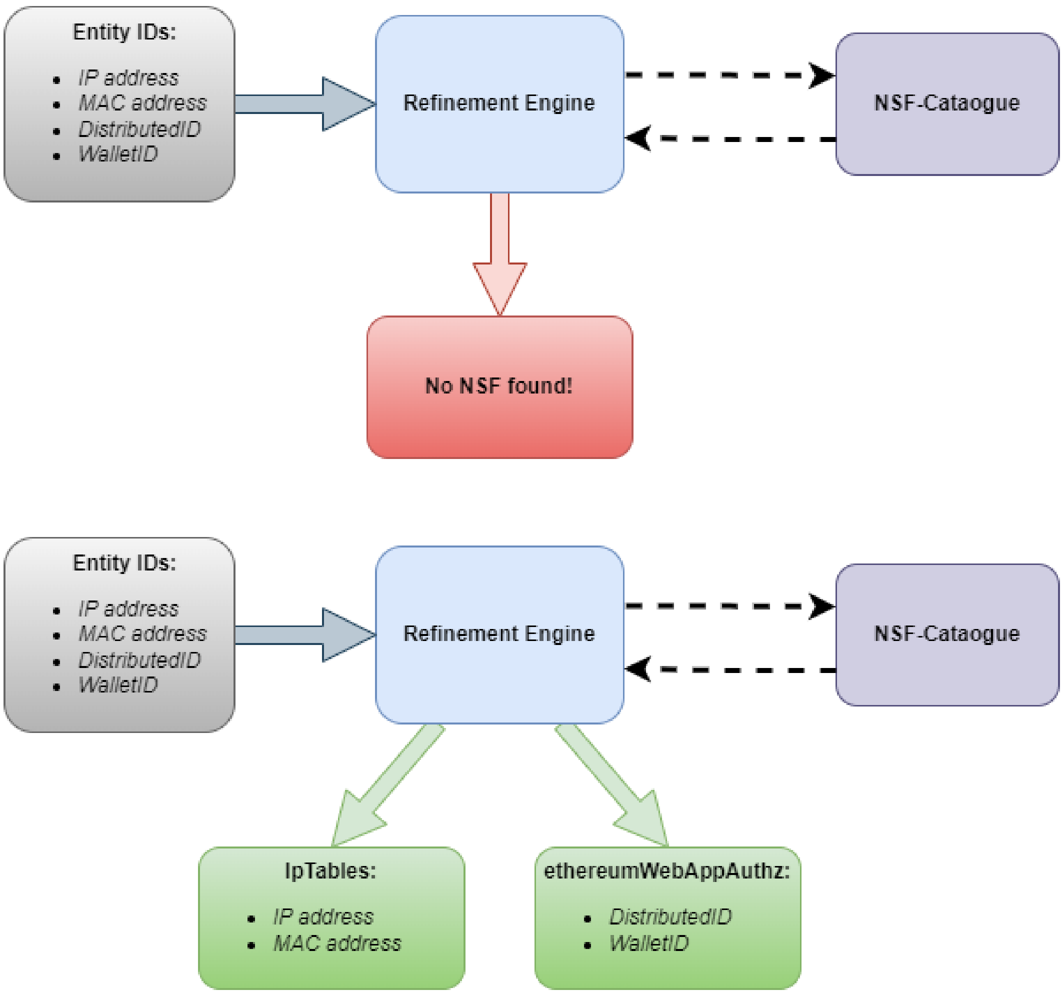


Figure 5.1: Before and after NSF-Catalogue querying strategy update

utilize intermediate information from previous refinements, enabling a dynamic evaluation of the current state compared to past data. By leveraging this stored knowledge, the system can optimize the refinement process, reducing redundant computations and ensuring that updates are both precise and efficient.

The essential design goal of the knowledge base is to support a systematic comparison between the current and the prior state of network policies and configurations. This comparison helps identify whether changes have occurred and, if so, their nature and extent. Based on this evaluation, the tool can adapt its behavior to process only the required updates, rather than reprocessing the entire configuration from scratch. This capability is especially useful when network environments and security requirements evolve incrementally.

The knowledge base supports the refinement tool to address three basic types of scenarios:

- *No change detected*: if the current state matches the previously processed state, the system can skip redundant refinement steps and reuse existing configurations;
- *Partial changes*: when some aspects of the policies or configurations are modified, the tool can focus on updating only the affected components, ensuring minimal reprocessing and improved efficiency;
- *Significant changes*: in cases where substantial differences are identified, the tool regenerates the necessary configurations, guaranteeing alignment with the new requirements.

This practice provides a further way of optimizing the refinement process while maintaining flexibility to handle both minor and major changes. By enabling dynamic adaptation, the knowledge base serves as a core component in modernizing the refinement tool, boosting the management of complex and evolving network environments and security requirements.

The following section will examine the taxonomy of possible scenarios concerning the high-level security policies managed through the use of a knowledge base, detailing the specific cases and their corresponding refinement.

5.3.1 HSPLs Taxonomy

This *taxonomy* categorizes situations based on the comparison between the current and previous states of HSPLs and their associated configurations. By proper differentiation, the refinement tool can implement suited strategies to optimize processing, reduce redundancy, and permit precise updates.

The taxonomy covers a spectrum of cases, ranging from complete matches where no updates are needed to situations requiring a full regeneration of configurations or their associated rules. Intermediate cases, where partial changes occur, are of particular relevance, as they represent opportunities to apply targeted updates while preserving unaffected components. These intermediate scenarios include distinct degrees of changes, like minor updates to specific attributes or significant shifts in network topology or security requirements. This structured classification balances efficiency with the need for accurate policy implementation.

A **new HSPL** is identified when its *ID* does not match any of those present in the knowledge base. In such cases, the strategy for handling it depends on a comparison with the conditions already recorded and known entities. New HSPLs are categorized into the following subcases:

- **No Match:**
  - When a new HSPL includes an *action* not already present among those previously recorded in the knowledge base

  - *Outcome:* both configuration and rule generation are required to implement the policy

- **Minor Changes:**
  - When the *action*, *subject*, and *object* match an already existing HSPL, even if the *options* are different
  - Alternatively, when the action remains the same, but the *subject* and/or *object* differ, as long as they are entities known in the knowledge base and maintain the same network paths
  - *Outcome*: neither configuration nor rule generation is required. Modifications are limited to updating the existing capabilities

- **Major Changes:**
  - When the *action* remains the same, but the *subject* and/or *object* change significantly, i.e.:
    * the *subject* and/or *object* are known entities in the knowledge base but have different network paths
    * the *subject* and/or *object* are not present in the knowledge base
  - *Outcome*: the configuration is required, then:
    * if the required capabilities and NSF remain the same, only the capability details need to be modified, and the selected device to configure may be replaced if necessary
    * if the required capabilities and/or NSF change, rule generation is necessary

An **existing HSPL** is identified when its *ID* matches one already present in the knowledge base. In these cases, the strategy depends on the degree of variation compared to the recorded policy. Existing HSPLs are classified into the following subcases:

- **Unchanged:**
  - When the *action*, *subject*, *object*, and *options* remain identical to those of the previously recorded HSPL
  - *Outcome*: no configuration or rule generation is required. The previous ones are maintained without modifications

- **Minor Changes:**
  - When the *action*, *subject*, and *object* are the same, but the *options* differ
  - Alternatively, when the *action* remains the same, but the *subject* and/or *object* differ, as long as they are entities known in the knowledge base and maintain the same network paths
  - *Outcome*: neither configuration nor rule generation is required. Updates are limited to modifying the existing capabilities

- **Major Changes:**
  - When the *action* remains the same, but the *subject* and/or *object* change significantly, i.e.:
    * the *subject* and/or *object* are known entities in the knowledge base but have different network paths
    * the *subject* and/or *object* are not present in the knowledge base
    * the *action* itself changes
  - *Outcome*: the configuration is required, then:
    * if the required capabilities and NSF remain the same, only the capability details need to be modified, and the selected device to configure may be replaced if necessary
    * if the required capabilities and/or NSF change, rule generation is necessary

A **removed HSPL** is identified when its *ID* is no longer present in the current list of HSPLs but exists in the knowledge base. This situation arises when a previously defined HSPL is not relevant or required anymore.

- **Removed:**
  - HSPLs recorded in the knowledge base but absent from the current list of HSPLs
  - *Outcome*: all information related to the HSPL, including configurations, associated entities, required capabilities, and options, is removed from the knowledge base. Additionally, any associated rules are deleted to maintain consistency

Conflict and Consistency Management

When applying the described taxonomy to handle new, existing, and removed HSPLs, it is essential to account for potential *conflicts*, *anomalies*, or *redundancies* that may arise during the refinement process. These issues can result from modifications, both minor and major, or the removal of HSPLs.

To ensure the integrity of the knowledge base and the correctness of the refinement outputs, additional checks must be performed, including:

- *Conflict detection*: verifying that changes in configurations or rules do not introduce contradictions with existing policies or system requirements;
- *Anomaly identification*: ensuring that updates or removals do not leave the system in an invalid or undefined state;
- *Redundancy elimination*: identifying and resolving duplicated or overlapping configurations or rules resulting from iterative modifications.

5.4 Code and Usability Improvements

In addition to the core design updates described in previous sections, further enhancements were introduced to improve the flexibility, maintainability, and usability of the refinement tool. These advancements focus on addressing limitations present in its earlier implementation.

Reducing Hardcoding

An important progress concerned eliminating hardcoded elements within the tool's codebase. In its past implementation, many parameters and configurations were directly embedded in the code, creating a rigid structure that made updates onerous and error-prone. To overcome this limitation, some parts of the refinement code were restructured to utilize dedicated *JSON* configuration files. This approach decouples the logic from the configuration, allowing the tool to dynamically retrieve necessary data from external files.

This shift not only enables configurations to be adjusted or extended without modifying the code itself, but also enhances maintainability by simplifying the codebase. Developers can now make adjustments to the tool's behavior by changing the *JSON files*, ensuring a more modular approach. Moreover, separating configuration data into external files reduces the risk of introducing bugs, supporting a more linear workflow.

Logging System

Another improvement was the integration of a robust *logging system*, designed to provide organized insights into the refinement process. The system captures every step of the workflow, categorizing events by severity, from simple debug to error and critical levels. This hierarchical approach guarantees that logs are both informative and manageable, permitting users to focus on the most relevant information for problem resolution or performance analysis.

The logging system records detailed information about the tool's actions and state changes, offering a complete overview of the workflow. At the same time, the console output was simplified to display only the essential steps, assuring clarity for the user while maintaining detailed logs in a separate file for in-depth analysis when needed.

Chapter 6

Implementation

This chapter provides a detailed examination of the implementation phase, focusing on the key elements discussed in the "*Design Overview*". It describes how the proposed solutions were translated into code, emphasizing the involved aspects of the refinement tool's architecture. Each section highlights the concrete realization of certain features, including data configuration and description files, as well as the scripts required for their processing and functions needed to enforce new mechanisms. Both newly introduced components and modifications to existing functionality are covered, offering a comprehensive view of the refinement tool's evolution.

Specifically, the chapter addresses:

- the implementation of the *TOSCA YAML-based network layout description*, detailing the parsing process, the integration of *JSON schemas* for validation, and the structures generated to support further refinement steps;

- the enhanced *querying mechanism for the NSF-Catalogue*, deepening the logic used to split security requirements in *combinations of NSFs*;

- the integration of a *knowledge base*, describing how state comparison is performed to optimize the policy refinement process and focusing on the actual strategies to face possible scenarios;

- additional operational improvements, i.e. the introduction of *JSON-based configuration dictionaries* to replace hardcoding and a comprehensive *logging system.*

Each of these topics is explored in dedicated sections, accompanied by code explanations and insights into the reasoning behind key implementation decisions.

6.1 TOSCA YAML Model Implementation

6.1.1 Type Definitions

To standardize the representation of network layouts, the *TOSCA Simple Profile in YAML Version 1.2* was adopted as a base. A dedicated file named `custom_types.yaml` defines custom node and capability types. These types extend the standard TOSCA definitions, matching them to meet the requirements of the refinement tool. The purpose and structure of possible node types included in this file are described below.

Node definitions

The `node_types` are derived from standard TOSCA types and further customized to include properties, capabilities, and requirements specific to the tool.

- **NetworkDevice**: an intermediate network device capable of implementing security functions, e.g. firewalls or VPN gateways.
  - *Properties*: includes an optional `id` property for device identification
  - *Capabilities*: includes *NSFs* to indicate supported security functions and standard `tosca.capabilities.Node` to permit connections with other nodes
  - *Requirements*: specifies connections to other devices or subnets (`DependsOn` or `LinksTo` relationships)
- **CustomSubnet**: a basic subnet with a customizable attribute for marking IP ranges as "*negated*".
  - *Properties*: derives the `cidr` property to specify the IP range and further includes a `negated` boolean to denote excluded address ranges
- **NetworkUser**: a regular network user or endpoint within a subnet.
  - *Properties*: includes attributes like the derived `ip_address` and, in addition, `mac_address`, `WID`, or `DID` for user identification
  - *Capabilities*: specifies *CustomEndpoint* to allow `url_path` as identifier and *NSFs* for supported security functions
  - *Requirements*: mandates at least one `LinksTo` connection to a subnet
- **MaliciousUser**: potential threats or adversarial entities, derived from the *CustomSubnet* type to reuse its negation feature.
  - *Properties*: similar to *NetworkUser*, with optional attributes for identification, but tracing IP ranges (`cidr`) instead of individual IP addresses, and without supporting *NSFs*

  - *Requirements*: connections to other devices, subnets, users, or web applications are allowed, properly using the `DependsOn`, `LinksTo`, or `ConnectsTo` relationship types

- **WebApp**: hosted applications and services within the network.
  - *Capabilities*: includes *NSFs* for security functions and *CustomEndpoint* for relating properties
  - *Requirements*: requires at least one `LinksTo` connection to a subnet

- **Traffic**: specific types of network traffic.
  - *Capabilities*: uses *CustomEndpoint* to describe traffic details like `protocol`, `port`, or `destination_type`

The defined custom node types derive from TOSCA standard types as follows: *NetworkDevice* and *Traffic* derive from `tosca.nodes.Root`, *CustomSubnet* derives from `tosca.nodes.network.Network`, and both *NetworkUser* and *WebApp* derive from `tosca.nodes.network.Port`. Additionally, *MaliciousUser* derives from the customized `company.nodes.CustomSubnet` to extend its functionalities.

Capability definitions

The `capability_types` section in the same file defines reusable capabilities for the nodes:

- **NSFs**: derives from `tosca.capabilities.Root` and represents the security functions a node can support. It contains a `supported_functions` property as a list, with each entry including the function name (`function`) and an associated `processing_order`.
- **CustomEndpoint**: extends the standard `tosca.capabilities.Endpoint` capability type. It defines properties like `protocol` (list of supported protocols) and `destination_type` (to specify the kind of destination), in addition to `port` and `url_path` derived properties.

By introducing these node and capability types, the tool achieves a modular and reusable structure for modeling diverse network layouts. Each type aims to address the unique attributes and relationships of the components it represents, ensuring both extensibility and compliance with the TOSCA standard and facilitating the process of defining new network descriptions.

6.1.2 Topology and Node Templates

The second file, `custom_templates.yaml`, complements the type definitions by describing the actual instances of nodes within the network. Each instance is based on the types defined in the imported `custom_types.yaml` file, with attributes and relationships suitable for reflecting the specific components and topology of the network.

The following examples illustrate some node instances that outline the usage of various custom types:

- **Firewall1 (NetworkDevice)**: represents a network device, specifically a firewall, identified by the *ID* “`firewall-1`”. This instance supports two security functions, `IpTables` and `XFRM`, to which a specific processing order is assigned. It connects to multiple subnets (`Subnet1.1` and `SubnetDMZ`) and to `Internet` through `LinksTo` relationships. Additionally, it is linked on another device, `Firewall2`, as indicated by a `DependsOn` relationship. These connections define its role and placement within the network topology (Listing 6.1).

```
  Firewall1:
    type: company.nodes.NetworkDevice
    properties:
      id: "firewall-1"
    capabilities:
      NSFs:
        properties:
          supported_functions:
            - function: "IpTables"
              processing_order: 1
            - function: "XFRM"
              processing_order: 2
    requirements:
    - device_subnet:
        node: Subnet1.1
        relationship: tosca.relationships.network.LinksTo
    - device_subnet:
        node: SubnetDMZ
        relationship: tosca.relationships.network.LinksTo
    - device_subnet:
        node: Internet
        relationship: tosca.relationships.network.LinksTo
    - device_device:
        node: Firewall2
        relationship: tosca.relationships.DependsOn
```

Listing 6.1: Firewall1 as a NetworkDevice

- **Subnet3.2 (CustomSubnet)**: models a standard subnet with a defined **IP address range** of “`10.3.2.0/24`”. As a *CustomSubnet*, it inherits the properties of a network subnet (`cidr`) and serves as a group of items within the network that can be linked to security devices (Listing 6.2).

```
  Subnet3.2:
    type: company.nodes.CustomSubnet
    properties:
      cidr: "10.3.2.0/24"
```

Listing 6.2: Subnet3.2 as a CustomSubnet

- **Internet (CustomSubnet)**: represents a specific use of the *CustomSubnet* type with the `negated` property set to `true`. This configuration marks the IP range "`10.0.0.0/8`" as "*negated*", which can be useful for modeling restricted access or external network exclusions (Listing 6.3).

```
  Internet:
    type: company.nodes.CustomSubnet
    properties:
      cidr: "10.0.0.0/8"
      negated: true
```

Listing 6.3: Internet as a CustomSubnet with negated flag

- **Bob_Endpoint (NetworkUser)**: plays a regular network user or endpoint connected to `Subnet1.1` through a mandatory `LinksTo` relationship. It is identified by its specific IP address "`10.1.1.12`". As a *NetworkUser*, this node depicts how individual users or endpoints are incorporated into the network topology and associated to a subnet (Listing 6.4).

```
  Bob_Endpoint:
    type: company.nodes.NetworkUser
    properties:
      ip_address: "10.1.1.12"
    requirements:
    - to_subnet:
        node: Subnet1.1
        relationship: tosca.relationships.network.LinksTo
```

Listing 6.4: Bob_Endpoint as a NetworkUser

- **Malicious_UserFULL (MaliciousUser)**: models a potentially malicious entity with both an IP range ("`192.168.0.0/30`") and a specific MAC address ("`123456789 abcdef0123456789abcdef01234`") for identification. It establishes a `ConnectsTo` relationship with a *WebApplication* node, `Web_App`, reflecting its role as a potential threat interacting with network resources (Listing 6.5).

```
  Malicious_UserFULL:
    type: company.nodes.MaliciousUser
    properties:
      cidr: "192.168.0.0/30"
      mac_address: "123456789abcdef0123456789abcdef01234"
    requirements:
    - to_webApp:
        node: Web_App
        relationship: tosca.relationships.ConnectsTo
```

Listing 6.5: Malicious_UserFULL as a MaliciousUser

- **Web_App (WebApp)**: represents a hosted web application within the network, identified by the IP address "`10.3.1.1`". It specifies additional properties such as a `url_path` "`www.webappsyn.com`", supported `protocol`(s) (`tcp` and `udp`), and a designated `port` (`9999`), illustrating the use of the *CustomEndpoint* capability. It also supports a security function, `PF-OpenBSD-PacketFilter`, specifying the `processing order`. Its connection to `Subnet3.1`, via a `LinksTo` relationship, locates it within the network infrastructure (Listing 6.6).

```
    Web_App:
      type: company.nodes.WebApp
      properties:
        ip_address: "10.3.1.1"
      capabilities:
        endpoint:
          properties:
            url_path: "www.webappsyn.com"
            protocol: ["tcp", "udp"]
            port: 9999
        NSFs:
          properties:
            supported_functions:
              - function: "PF-OpenBSD-PacketFilter"
                processing_order: 1
      requirements:
      - to_subnet:
          node: Subnet3.1
          relationship: tosca.relationships.network.LinksTo
```

Listing 6.6: Web_App as a WebApp

- **DNS traffic (Traffic)**: acts a specific type of network traffic characterized by its `destination_type` (`DNS`), `protocol` (`udp`), and `port` (`53`). As a *Traffic* node, it utilizes the *CustomEndpoint* capability to define these attributes, serving as an example of how traffic types are modeled within the network layout (Listing 6.7).

```
    DNS traffic:
      type: company.nodes.Traffic
      capabilities:
        endpoint:
          properties:
            destination_type: DNS
            protocol: [udp]
            port: 53
```

Listing 6.7: DNS traffic as a Traffic

6.1.3 JSON Validation Schemas

To ensure the correctness and consistency of the *TOSCA YAML files*, two *JSON schema files*, `types_schema.json` and `template_schema.json`, were created to validate the structure of the `custom_types.yaml` and `custom_templates.yaml` files, respectively. They impose a strict set of rules, defining allowed properties, capabilities, and relationships, as well as permissible values for various attributes, helping to automate error detection during file editing.

Types JSON schema

This schema validates the `custom_types.yaml` file, ensuring the proper definition of node and capability types. Key validation rules include:

- *General constraints*:
  - the *TOSCA version* must be `tosca_simple_yaml_1_2`
  - *node* types must be prefixed with "`company.nodes.`", and *capability* types with "`company.capabilities.`"
- *Node types*:
  - must derive from provided *standard TOSCA* types (`tosca.nodes.Root`, `tosca.nodes.network.Network`, `tosca.nodes.network.Port`) or other *custom* types
  - for each *property* must be specified its `type`. It can optionally be reported if they are `required` or have a `default` value
  - *requirements* must include a valid `capability` for the type of pointed `node` and an allowed type of `relationship` (`DependsOn`, `LinksTo`, `ConnectsTo`). It can also be set limitations to their `occurrences`
- *Capability types*:
  - *properties* must define a valid `type`, and can both include specifications about `entry_schema` for complex data structures and if they are `required`

Templates JSON schema

This schema dynamically checks the `custom_templates.yaml` file, guaranteeing that node templates conform to the proper way of defining them. Main real-time validations concern:

- *General constraints*: the *TOSCA version* must be `tosca_simple_yaml_1_2`. In addition, the file must include an `imports` section referencing the `custom_types.yaml` file
- *Type reference*: each node must have a `type` attribute that matches the pattern `"^company.nodes..*"`, ensuring that custom-defined node types are used

- *Properties validation*: properties can be of any supported data type (`string`, `number`, `boolean`, `array`, or `object`)
- *Capabilities validation*: capabilities, if present, must include the capability `type`, with the correct internal structure, as explicitly defined in the schema. In the case of `NSFs` capability, only predefined `supported_functions` are allowed, also specifying the `processing_order` for each of them
- *Requirements validation*: requirements must define the referenced `node` by name and a type of `relationship` between supported ones (`DependsOn`, `LinksTo`, or `ConnectsTo`)

Associating schemas with YAML files

The discussed *JSON schemas* are associated with their respective *TOSCA YAML files* using the `settings.json` configuration file. This file ensures that any *YAML editor* supporting schema validation emphasizes errors directly during file editing.

The configuration associates `template_schema.json` with files named `*_templates.yaml` and `types_schema.json` with files named `*_types.yaml` (Listing 6.8).

```
    "yaml.schemas": {
    "./Yaml_Schemes/template_schema.json": "*_templates.yaml",
    "./Yaml_Schemes/types_schema.json": "*_types.yaml"
  }
```

Listing 6.8: Schemas association in `settings.json`

Retrieving NSFs from catalogue

The `NSFCatalogue.xml` file serves as a repository of supported *Network Security Functions*, organized as external entities. These NSFs are referenced in *TOSCA node templates* to define supported security functionalities. Parsing this file ensures that the *JSON validation schema* remains up-to-date and consistent with the latest NSF definitions.

The XML file lists NSFs entities through external references, making it easy to maintain and extend (Listing 6.9).

```
...

<!DOCTYPE p:nsfCatalogue [
  <!ENTITY securityCapabilities     SYSTEM "SecurityCapabilities.xml">
  <!ENTITY genericPacketFilter      SYSTEM "genericPacketFilter.xml">
  <!ENTITY IpTables                 SYSTEM "IpTables.xml">
  <!ENTITY PF-OpenBSD-PacketFilter  SYSTEM "PF-OpenBSD-PacketFilter.xml
    ">
  <!ENTITY sonaeOperatorOutput      SYSTEM "sonaeOperatorOutput.xml">
  <!ENTITY Squid                    SYSTEM "Squid.xml">
  <!ENTITY StrongSwan               SYSTEM "StrongSwan.xml">
  <!ENTITY XFRM                     SYSTEM "XFRM.xml">
]>

<p:nsfCatalogue ...>

    <!-- List of nSF entities and their capabilityTranslationDetails -->
    &IpTables;
    &PF-OpenBSD-PacketFilter;
    &Squid;
    &StrongSwan;
    &XFRM;

    ...

</p:nsfCatalogue>
```

Listing 6.9: `NSFCatalogue.xml` fragment

The parsing script `NSFCatalogue_parsing.py` manages the extraction and integration of these entities into the validation process:

- the `parse_xml_entities(xml_file)` function reads the XML file, extracts all NSF entities, and filters only those explicitly used in the catalogue. This ensures irrelevant entities are excluded, maintaining a clean and accurate list of available NSFs. By leveraging regular expressions, the script identifies entity names from `<!ENTITY>` declarations and verifies their usage (e.g. `&IpTables`);
- once the entities are extracted, the `update_json_schema(json_file, nsf_entities)` function updates the `template_schema.json` file. This function modifies the *enum* field for the `function` property within the `NSFs` capability, replacing it with the extracted NSF list. This update ensures that only predefined NSFs are considered valid during the TOSCA `node_templates` editing.

6.1.4 Parsing Scripts

The parsing process is handled by two interdependent *Python scripts*: `TOSCA_types_parser.py` and `TOSCA_layout_parser.py`. Each script satisfies a distinct purpose within the overall workflow. The first one focuses on processing *node* and *capability types* to dynamically define Python data structures through the `dataclasses` library, while the second one parses *node templates* and *topology* to create instances of the generated dataclasses, providing dictionaries of the various components and building a network graph reflecting the topology. Together, these scripts serve to link the *TOSCA YAML model* to the *operational refinement logic.*

Parsing type definitions

The `TOSCA_types_parser.py` script is responsible for processing the `custom_types.yaml` file to define Python dataclasses on-the-fly for the node and capability types described in the TOSCA YAML model. The primary function in this script is `parse_definitions()`, which orchestrates the parsing workflow. Supporting functions handle tasks like loading the TOSCA YAML file, collecting both custom and inherited attributes, and creating dynamic classes. In detail:

- `load_tosca_template(filename)`

  reads the TOSCA YAML file and initializes a `ToscaTemplate` object to provide access to its structured data;

- `collect_properties_and_capabilities(node_type_name, ...)`

  recursively gathers properties and capabilities from a node type and its base types (using internal `collect_from_base_type(base_type_name)` function), supporting inheritance;

- `create_dynamic_dataclass(name, properties, ...)`

  dynamically defines Python dataclasses based on the properties and capabilities collected for each node type;

- `parse_definitions(filename)`

  coordinates the parsing process (Listing 6.10):

  – loads the TOSCA template using `load_tosca_template()`;
  – collects attributes for each node type using `collect_properties_and_capabilities()`;
  – dynamically creates dataclasses using `create_dynamic_dataclass()` and stores them in a dictionary for later use.

```python
def parse_definitions(filename):
    tosca_template = load_tosca_template(filename)
    dynamic_classes = {}

    capabilities = tosca_template.tpl.get('capability_types', {})
    node_types = tosca_template.tpl.get('node_types', {})

    for node_type_name in node_types:
        properties, relevant_capabilities =
        collect_properties_and_capabilities(node_type_name, node_types,
        capabilities, standard_properties)

        dynamic_class = create_dynamic_dataclass(node_type_name,
        properties, relevant_capabilities, standard_capabilities)

        dynamic_classes[node_type_name] = dynamic_class

    return dynamic_classes
```

Listing 6.10: Function `parse_definitions()`

Parsing network layout

The `TOSCA_layout_parser.py` script processes the `custom_templates.yaml` file to create instances of the *dataclasses* defined by the other parsing script. The main function, `get_network_data()`, coordinates the workflow to instantiate dataclasses, categorize elements into specific dictionaries, and represent the network topology. Specifically:

- `create_instance(node_type, properties, ...)`

  instantiates a dataclass for a specific node type, populating its fields with values retrieved from the TOSCA YAML file;

- `categorize_nodes(tosca_template, node_categories)`

  populates a dictionary with node names and their categories based on their types;

- `filter_fields(obj, exclude_fields=None)`

  filters out unnecessary fields from the dataclass instances for cleaner output;

- `get_network_data(definitions_file, template_file)`

  organizes the parsing flow (Listing 6.11):

  - calls `parse_definitions()` function, imported from `TOSCA_types_parser.py`, to obtain dynamic dataclasses for the node types;
  - loads the TOSCA template using the `ToscaTemplate` class;
  - categorizes nodes into *subnets*, *devices*, and *entities*;
  - instantiates dataclasses for each `node_template` using `create_instance()`;
  - builds a visualizable *network graph* from the node `requirements` using the `NetworkX` library.

```python
def get_network_data(definitions_file, template_file):
    dynamic_classes = parse_definitions(definitions_file)

    try:
        tosca = ToscaTemplate(template_file)
        subnets, devices, entities = {}, {}, {}
        connections = NetworkBackbone()
        ...

        for node in tosca.nodetemplates:
            properties = {prop.name: prop.value for prop in
                            node.get_properties_objects()}
            capabilities = {cap.name: {prop.name: prop.value for prop in
                            cap.get_properties_objects()} for cap in
                            node.get_capabilities_objects()}
            instance = create_instance(node.type, properties,
                            capabilities, dynamic_classes)

            if 'subnet' in node.type.lower():
                subnets[node.name] = instance
            elif 'device' in node.type.lower():
                devices[node.name] = instance
            else:
                entities[node.name] = instance

            if node.requirements():
                ...
                for requirement in node.requirements:
                    for key, value in requirement.items():
                        if isinstance(value, dict) and 'node' in value
                                and 'relationship' in value:
                            ...
                            connections.graph.add_edge(node.name,
                                                    value['node'])
    ...
    return subnets, devices, entities, connections
```

Listing 6.11: Key segments of `get_network_data()` function

6.1.5 Adapting Refinement Code

The function `get_network_data()` from the `TOSCA_layout_parser.py` script is imported into the `refinement.py` script, which implements the logic of the refinement process. This function is invoked to extract the categorized dictionaries for `subnets`, `devices`, and `entities`, as well as the `connections` graph representing the topology. This integration ensures structured access to network layout information, improving the refinement operations' clarity and efficiency.

With this change, key functions in the refinement logic involved in manipulating network data, including `database_search_entity_info(x, hsplid)`, `entity_analysis(entity_string, hsplid)`, `add_entity_req_capabilities(hsplid, entity_string)`, `get_device_nsf_list(device)`, were updated to handle the structured dictionaries retrieved from calling `get_network_data()`. The use of these dictionaries facilitates the management of network data, enabling more precise and organized processing.

Furthermore, the previous `build_graph()` function, which was responsible for constructing the network graph, is no longer needed. The graph is directly supplied by the parsing script, eliminating the necessity for additional graph construction.

To provide compliance with the *TOSCA properties* nomenclature, the definition of the *CLIPS template* for `entity` was revised as shown below (Listing 6.12):

```
    (deftemplate entity
        (slot name (type STRING))
        (slot ip_address (type STRING))
        (slot WID (type STRING))
        (slot DID (type STRING))
        (slot mac_address (type STRING))
        (slot url_path (type STRING))
    )
```

Listing 6.12: Updated CLIPS template for `entity`

6.2 Enhanced NSFs Querying Mechanism Enforcement

To address the limitations of the previous querying approach, a new function, `find_nsfs_covering_caps(capabilities)`, was implemented in the `refinement.py` script to identify *combinations of NSFs* that collectively satisfy a given *set of capabilities*. This function integrates with the existing querying mechanism, `get_nsfs_with_cap(capabilities)`, which identifies NSFs that individually satisfy the full set of requested capabilities. If no such NSF is found, the new function expands the search by determining combinations of NSFs to cover all required capabilities.

6.2.1 Helper Functions Overview

Initially, the new function calls `get_nsfs_with_cap(capabilities)` to query the *NSF-Catalogue* via the *API* `capa_set_search.xq`, retrieving NSFs that independently support all specified capabilities. If successful, the process terminates early, and these NSFs are returned as a result.

If no single NSF can satisfy the requirements, the function proceeds with a more comprehensive strategy to find suitable combinations of NSFs, operating through three helper functions defined internally, each responsible for a distinct step of the process:

1. `build_capability_to_nsf_map(capabilities)`

   creates a dictionary where each capability is linked to a list of NSFs that support it. It first checks whether the required capability has already been queried, leveraging a caching mechanism to avoid redundant requests. If so, the function retrieves the corresponding NSFs directly from the cache. Alternatively, it calls the `get_nsfs_with_cap(capabilities)` function, passing a unique capability to fetch the relevant NSFs from the NSF-Catalogue and caching the result to optimize future queries.

2. `build_nsf_capabilities_dict(capability_to_nsfs)`

   using the capability-to-NSFs mapping generated in the previous step, derives a reverse dictionary that associates each NSF with the set of capabilities it can support. For every capability and its corresponding list of NSFs, the function iterates through the NSFs and updates the dictionary. If an NSF already exists in the dictionary, the new capability is added to its set of supported capabilities. If not, a new entry is created for the NSF. This step is essential for quickly identifying the coverage provided by each NSF during the combination search process.

3. `find_all_combinations(remaining_caps, current_combination, ...)`
   explores all possible combinations of NSFs that together cover the required capabilities. When there are no more capabilities to cover, the current combination is finalized, performing a check against duplicates and adding it to the list of all combinations. For each NSF in the `nsf_dictionary`, the function calculates its contribution to covering the remaining capabilities. A minimum threshold is applied, e.g. an NSF must cover at least 2 capabilities to be added in a combination. For each valid NSF, the function is called recursively with updated arguments: the reduced set of remaining capabilities, the current combination including the evaluated NSF, and the updated coverage. This step examines all potential solutions, prioritizing those with minimal redundancy.

6.2.2 Workflow Coordination

The overall workflow of `find_nsfs_covering_caps(capabilities)` begins with a direct query for NSFs that satisfy all requested capabilities by themselves. If such NSFs are found, they are returned immediately. Otherwise, the function sequentially executes the three steps above to generate combinations of NSFs (Listing 6.13).

The function returns:

- a *list of NSF combinations*: each combination is a group of NSFs that collectively fulfil the requirements and the list is ordered prioritizing combinations with fewer NSFs;
- an *NSF-to-capabilities mapping*: this aids in understanding the role of each NSF in satisfying the requirements.

```python
def find_nsfs_covering_caps(capabilities):
    ...
    initial_nsfs = get_nsfs_with_cap(capabilities)
    if initial_nsfs:
        ...
        for nsf in initial_nsfs:
            nsf_dictionary[nsf] = set(capabilities)
        return [[nsf] for nsf in initial_nsfs], nsf_dictionary

    capability_to_nsfs = build_capability_to_nsf_map(capabilities)

    build_nsf_capabilities_dict(capability_to_nsfs)

    find_all_combinations(set(capabilities), [], set())

    all_combinations.sort(key=len)

    return all_combinations, nsf_dictionary
```

Listing 6.13: Key segments of `find_nsfs_covering_caps()` function

The new `find_nsfs_covering_caps (capabilities)` is now invoked rather than `get_nsfs_with_cap(capabilities)` when NSF-Catalogue querying is required.

6.3 Operational Knowledge Base Management

6.3.1 Knowledge Base Structure

The knowledge base serves as a repository of intermediate information generated during the refinement process. It stores details about previously processed *HSPLs*, their associated *configurations*, and concerning *entities*, enabling the refinement tool to compare the current state with prior executions.

The structure of the relating file, `knowledge_base.json`, is organized into several key fields, retrieved from *facts* asserted in the *CLIPS environment* (Listing 6.14):

- *HSPLs*: include details about each security policy, as specified in the `HSPL.xml` file used to describe high-level policies, like its unique `id`, `subject`, `action`, and `object`, plus a *flag* indicating whether it requires configuration;
- *options*: specify additional attributes related to HSPLs, like time periods or other policy-specific parameters;
- *requested capabilities*: map each security capability required for implementation to the specific HSPL, including eventual details;
- *entities*: maintain information about the network entities involved in the policies, including their names and other identifying attributes, like `ip_addresses` referencing both single IP addresses and IP ranges;
- *notifications*: provide alerts or warnings generated during the refinement process;
- *configurations*: tracks the devices, along with their suitable NSFs, chosen for each HSPL during the configuration phase.

```
{
    "hspl": [...
        {"id": "hspl3", "subject": "Bob_Endpoint",
        "action": "protect integrity", "object": "Alice_Endpoint",
        "needs_configuration": "FALSE"},
    ...],
    "option": [...
        {"type": "time period", "value": "19:30 20:00",
        "hsplid": "hspl2"},
    ...],
    "reqcapability": [...
        {"capability": "IpDestinationAddressConditionCapability",
        "detail": "~10.0.0.0/8", "hsplid": "hspl4"},
    ...],
    "entity": [...
        {"name": "Web_App", "ip_address": "10.3.1.1", "WID": "",
        "DID": "", "mac_address": "", "url_path": "www.webappsyn.com"},
    ...],
    "notification": [...
        {"message": "This IP address is indicated as subject/object
        for the HSPL:", "detail": "172.16.1.16", "hsplid": "hspl1"}
    ...],
    "configuration": [...
        {"device": "Firewall1", "nsf": "IpTables", "hsplid": "hspl4"}
    ... ]
}
```

Listing 6.14: `knowledge_base.json` dictionary fields

In addition, the `intermediate_paths.json` file takes part in the refinement strategy, allowing for the storage of path information about entities involved in each policy. In this way, it plays a key role in topology-based comparisons, being useful to verify if changes to entities in the policies possibly maintain the same intermediate paths, and thus supporting further optimization of the refinement process. Just like the `knowledge_base.json`, it is updated during every execution to ensure it reflects the current network state.

6.3.2 New Functions Overview

The knowledge base integration into the refinement process required the implementation of certain functions within the `refinement.py` to enable both comparison and update of HSPLs and their associated configurations and rules. These functions collectively guarantee that the refinement tool can actually handle differences between the current and previous states. Each of these functions is detailed below, explaining their roles and workflows across the overall refinement strategy.

- `compare_hspls(current_hspl, optional_fields, ...)`

  is responsible for comparing a single *current HSPL* against those stored in the previous knowledge base to categorize it based on detected differences. The workflow involves:

  – if the `action` of `current_hspl` matches with an HSPL retrieved from `previous_knowledge_base` using the same `id`, it will be collected within `hspls_state['existing']`, after further checks are performed:

    ∗ if the `subject`, `object`, and `optional_fields` are identical, the HSPL is classified as "*unchanged*";

    ∗ if the `subject` and `object` are the same but the options differ, it is classified as having "*minor changes*";

    ∗ if either the `subject` or `object` differs, network paths between the entities are compared using `paths_dict` argument and `path_search(...)` function. A match results in a "*minor changes*" classification, while mismatched paths lead to "*major changes*", setting `needs_configuration` to *True*;

  - vice versa, if the `action` does not match, the HSPL is categorized as *"existing"* but having "*major changes*", requiring configuration updates;
  - if no HSPL with the same `id` exists in the previous knowledge base, a scoring mechanism (`match_score`) is used to evaluate similarities in `action`, `subject`, `object`, and network paths. Depending on the score, the HSPL is categorized into "*minor changes*", "*major changes*", or "*no match*" within `hspls_state['new']`.

- `modify_existing_rules(configs, current_req_cap, ...)`

  updates *existing rules* for an HSPL effected by *minor changes*, by comparing the currently required capabilities (`current_req_cap`) with those previously required (`prev_req_cap`). The goal is to preserve consistent configurations without fully regenerating the rules. The steps are:

  - previous requested capabilities are mapped into `prev_cap_map` and current ones into `current_cap_map` for comparison, in order to identify:
    * *modified capabilities*, where the current `detail` field differs from the previous one. These are tracked into `modified_capabilities` for subsequent checks;
    * *unchanged capabilities*, where previous and current `detail` fields are equal;
    * *new capabilities*, when absent in `prev_cap_map`;

    each of these capabilities is added to `updated_capabilities`;
  - `previous_rules` are examined to maintain those specific capabilities included during original rules generation, checking against modifications and duplicates. These capabilities are added to `total_capabilities` together with capabilities retained in `updated_capabilities`;
  - relevant capabilities involving source and destination addresses are swapped using `swap_cap_src_dest(...)` to take into account bidirectional rules, when necessary;
  - the `updated_rules`, obtained for each configuration in the `configs` argument, are then returned by the function.

- `recycle_previous_rules(configs, current_req_cap, ...)`

  This function attempts to reuse *existing rules* when *major changes* do not necessitate full regeneration, but rather adjustments to existing configurations. The process consists of:

  - first checking if the current and previous sets of required capabilities (`current_capabilities_names` and `prev_capabilities_names`) match. If they do, also `current_nsf` and `prev_nsf` are verified to be the same, for each configuration in the `configs` argument. If so, the function proceeds to adjust existing rules;
  - the details of matching capabilities are replaced with the actual ones to meet the current requirements, ensuring duplicates do not occur;
  - capabilities' source and destination are properly adjusted for each rule through `swap_cap_src_dest(...)` to handle bidirectionality;

  - the `recycled_rule` are aggregated, possibly changing the configured `device`, and returned by the function.

- `update_knowledge_base(facts_dict, previous_knowledge_base, ...)`

  updates the `knowledge_base.json` file to reflect the latest configurations, incorporating both existing HSPLs and newly identified ones. The main actions are:

  - starting from the current dictionary of facts asserted in the CLIPS environment (`facts_dict`), keeping track of novel configurations for both *new* and *existing HSPLs* with "*major changes*" and those having "*no match*" with any of the previous ones;
  - configurations for *existing HSPLs* categorized as "*unchanged*" or having "*minor changes*" are retained from `previous_knowledge_base`. For *new HSPLs* with "*minor changes*", configurations from the best-matching previous HSPLs are copied and updated with the `new_hspl_id`;
  - configurations for HSPLs marked as "*removed*" are excluded;
  - the `updated_knowledge_base` is written back to the JSON file specified by `kb_filename`.

- `update_path_data(file_path, hspl_states)`

  manages the path information stored in `intermediate_paths.json`, ensuring consistency with the latest refinement process. The procedure includes:

  - for *new HSPLs* with "*minor changes*", path `data` from similar HSPLs is duplicated and updated with the HSPL's `new_id`;
  - entries corresponding to "*removed*" HSPLs are deleted;
  - the updated path `data`, together with unchanged paths and those already reported during the last configuration phase, is saved back to the indicated `file_path`.

6.3.3 New Refinement Strategy

The `main()` function now orchestrates the refinement process by coordinating multiple steps to analyze current and previous knowledge bases, categorize HSPLs, and manage intermediate rule updates or generation. Below, the new workflow is presented, divided into four consecutive phases, each illustrated with relevant code snippets.

Loading information

In this phase, the `main()` function initializes needed data structures and loads the necessary information from the previous execution. These include the *previous knowledge base*, *rules*, and *intermediate paths*, which are used to track changes and maintain coherence across refinements.

```
    ...
    previous_knowledge_base = {}
    previous_rules = []
    previous_paths = {}

    hspl_states = { 'new': {'no_match': [], 'minor_changes': [],
    'major_changes': []},  'existing': {'minor_changes': [],
    'major_changes': [], 'unchanged': []},  'removed': {'removed': []} }

    try:
        with open(knowledge_base_filename, 'r') as prev_file:
            previous_knowledge_base = json.load(prev_file)
            ...
    try:
        with open(output_filename, 'r') as output_file:
            previous_rules = json.load(output_file)
            ...
    try:
        with open(intermediate_paths_filename, 'r') as
        intermediate_paths_file:
            previous_paths = json.load(intermediate_paths_file)
            ...
```

Listing 6.15: Initializing data structures and loading previous information

The code displayed (Listing 6.15) initializes `hspl_states`, a dictionary that categorizes the current HSPLs into states (basically `new` and `existing`), collecting `removed` ones too. It then attempts to load the knowledge base, rules, and paths from *JSON files*, proceeding with HSPLs comparison in case of successful loading. Alternatively, the refinement process will start configuration and rule generation from scratch.

Comparing HSPLs

The next step is to parse the current HSPL file (`policy_filename`), extract individual HSPLs, and compare them with previously stored ones. Each HSPL is classified based on its differences from the `previous_knowledge_base`.

```
    ...
    file = et.parse(policy_filename)
    hspl_list = file.getroot()
    current_hspls_ids = set()
    for hspl in hspl_list.iterfind('.//{http://fishy-project.eu/hspl
        }hspl'):
        parsed, optional_fields = parse_hspl(hspl)
        current_hspls_ids.add(parsed['id'])
        needs_config = compare_hspls(parsed, optional_fields,
                        previous_knowledge_base, hspl_states,
                        previous_paths)
        parsed['needs_configuration'] = clips.Symbol("TRUE" if
                                        needs_config else "FALSE")
        ...
    previous_hspls_ids = set(hspl['id'] for hspl in
                            previous_knowledge_base.get('hspl', []))

    hspl_states['removed']['removed'] = list(previous_hspls_ids
                                               - current_hspls_ids)
    ...
```

Listing 6.16: Comparing current and previous HSPLs

Here (Listing 6.16), the `compare_hspls(...)` function is used to categorize each `parsed` HSPL into different lists ("*no match*", "*minor changes*", "*major changes*", or "*unchanged*"), filling the `hspl_states` dictionary accordingly. "`removed`" HSPLs are identified by comparing the `previous_hspls_ids` and `current_hspls_ids`.

During this phase, the `needs_config` flag, based on the value returned by the `compare_hspls(...)` function, is attached as a field into the `parsed` HSPLs, which are then asserted into the CLIPS environment. Its boolean value is subsequently used in the CLIPS rule engine to discriminate HSPLs requiring configuration processing. In this regard, the *CLIPS rule* `path-analysis` was modified to explicitly check this flag to decide whether to invoke the `find-configuration` function (Listing 6.17).

```
    (defrule path-analysis
        (declare (salience -1))
        (hspl    (id ?id)
                 (subject ?sub)
                 (action ?act)
                 (object ?obj)
                 (needs_configuration TRUE))
        (entity  (name ?sub)
                 (ip_address ?ips))
        (entity  (name ?obj)
                 (ip_address ?ipd))
        =>
            (find-configuration ?ips ?ipd ?id)
    )
```

Listing 6.17: Updated `path-analysis` CLIPS rule

Managing cases

At this point, the categorized HSPLs are processed to determine the appropriate action for each case. Depending on their state, HSPLs may trigger *updates*, *recycling of existing rules*, or the *generation of new rules*.

```
...
for state, changes in hspl_states.items():
    for change_type, hspls in changes.items():
        for hspl in hspls:

            if state == 'new' and change_type != 'no_match': ...
                if change_type == 'minor_changes':
                    modified_rule = modify_existing_rules(prev_configs,
                                req_cap,prev_req_cap,hspl_rules,hsplid)
                    intermediate += modified_rule
                elif change_type == 'major_changes': ...
                    recycled_rules = recycle_previous_rules(
                            current_configs,req_cap, prev_req_cap,
                                            hspl_rules,hsplid)
                    if recycled_rules:
                        intermediate += recycled_rules
                    else:
                        for conf in current_configs: ...
                            intermediate += generate_rules(conf,req_cap)

            elif state == 'existing' and change_type != 'unchanged': ...
                if change_type == 'minor_changes':
                    modified_rule = modify_existing_rules(prev_configs,
                                req_cap,prev_req_cap,hspl_rules,hsplid)
                    intermediate += modified_rule
                elif change_type == 'major_changes': ...
                    recycled_rules = recycle_previous_rules(
                            current_configs,req_cap,prev_req_cap,
                                            hspl_rules,hsplid)
                    if recycled_rules:
                        intermediate += recycled_rules
                    else:
                        for conf in current_configs:
                            intermediate += generate_rules(conf,req_cap)

            elif change_type == 'no_match':
                for configuration in facts_dict['configuration']:
                    if hspl == configuration['hsplid']: ...
                        intermediate += generate_rules(configuration,
                                                            req_cap)

            elif change_type == 'unchanged':
                unchanged_rules = list(filter(lambda r: r['hsplid'] ==
                                                hspl, previous_rules))
                intermediate.extend(unchanged_rules)
...
```

Listing 6.18: Managing HSPL states

This snippet (Listing 6.18) shows how the function handles HSPLs based on their state (`new`, `existing`, `removed`) and subcategory (`minor_changes`, `major_changes`, `no_match`, `unchanged`). Introduced functions (`modify_existing_rules(...)` and `recycle_previous_rules(...)`) are invoked to update or try to recycle `intermediate` rules, otherwise a call to the existing `generate_rules(...)` is performed to generate them from scratch.

Updating information for future refinements

Finally, the *knowledge base* and *intermediate paths* are updated to reflect the latest refinement, being reliable for the next execution. The process involves saving updated HSPLs' details, configurations, and paths while removing obsolete entries.

```
...
update_knowledge_base(facts_dict,previous_knowledge_base,
                        hspl_states,knowledge_base_filename)

update_path_data(intermediate_paths_filename,hspl_states)

if output_filename:
    json.dump(intermediate, open(output_filename, 'w'), indent=4)
...
```

Listing 6.19: Updating knowledge base and intermediate paths

Here (Listing 6.19), the `update_knowledge_base(...)` function integrates the new configurations based on the categorized HSPLs' states, paying attention that those marked as "*removed*" are excluded. The `update_path_data(...)` function aligns the intermediate paths with the current refinement, discarding "*removed*" ones also in this case. In conclusion, the refined `intermediate` rules are saved into the specified `output_file`.

6.4 Hardcoding Removal and Logging Integration

This section highlights two minor but notable improvements to the implementation: removing hardcoded values and integrating a logging system.

6.4.1 Removing Hardcoding through Configuration Dictionaries

To boost adaptability and support maintenance, previously hardcoded values have been replaced with dynamically loaded *configurations* stored in *JSON files*. They are located in the `src/config/` directory, aiming to centralize various settings and mappings needed during the refinement process.

This information is now retrieved on demand using the following pattern (Listing 6.20):

```
with open('src/config/environment_config.json') as config_file:
    config = json.load(config_file)
```

Listing 6.20: Example of configuration loading from `file.json`

These `config` dictionaries allow no modification to the source code when certain static updates are required.

6.4.2 Logging System Configuration and Usage

To replace the previous reliance on standard output, a *logging system* has been integrated using Python's `logging` module.

A dedicated `logger_config.py` file in the `src/logger/` directory defines the logging setup. It configures a "*logger*" to capture messages at different levels and outputs them to both a `refinement.log` file and the *console*. The `file_handler` logs all messages, including `DEBUG`, while the `console_handler` limits output to `INFO` and higher levels, keeping on the standard output a basic understanding of the workflow. In addition, log formatting includes *timestamps*, *logger names*, *levels*, and *messages* for clarity.

The `logger` object is used throughout `refinement.py` for consistent logging at different levels, for instance (Listing 6.21):

```
    logger.debug('REQUIRED CAPABILITIES: %s', req_capabilities)

    logger.info("Previous knowledge base upload done.")

    logger.error('Something went wrong: errors detected or configuration
                    issues: %s', facts_dict.get('error'), exc_info=True)
```

Listing 6.21: Examples of `logger` object usage

Chapter 7

Validation and Testing

The chapter provides the validation and testing of the proposed advancements, demonstrating how the implemented strategies meet the prefixed objectives. The work is evaluated across three core aspects representing the significant contributions of this thesis:

- testing the correct parsing, processing, and utilization of the *TOSCA YAML files* to define network layouts, evidencing the formalization of the approach;
- validating the ability to distribute *security capabilities* across *multiple NSFs* when a single NSF cannot satisfy all the required capabilities, expanding the application scope of policy enforcement;
- assessing the accuracy of the optimized refinement process by leveraging stored *intermediate information*, verifying updated configurations.

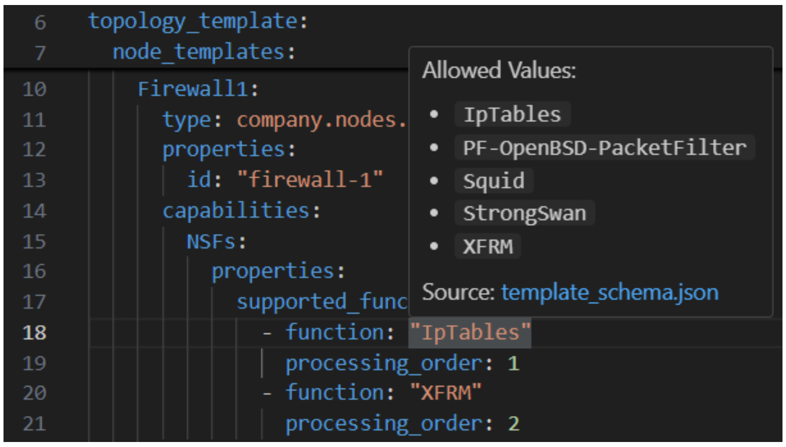
For each section, concrete results obtained during testing are illustrated, attesting how novel developments achieve their intended goals by addressing both functional correctness and practical usability.

7.1 TOSCA YAML Model Results

7.1.1 Real-time Validation of TOSCA YAML Files

The integration of *JSON schemas* for *type* and *template* validation allows real-time feedback during the editing process. The following images illustrate some examples of warnings supplied by the validation mechanism when attempting to edit improperly defined *TOSCA YAML files*:

- The example in Figure 7.1 demonstrates how the `template_schema.json` ensures that the *NSFs* assigned to the `capabilities` property of a node are limited to the pre-defined set of values. Here, the editor highlights a list of valid values for the `function` field under the `supported_functions` property.

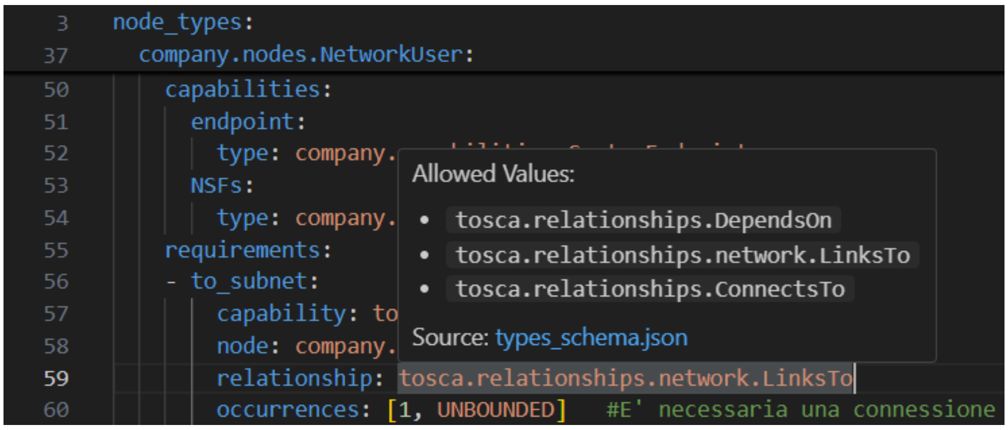


```
topology_template:
  node_templates:
    Firewall1:
      type: company.nodes.
      properties:
        id: "firewall-1"
      capabilities:
        NSFs:
          properties:
            supported_func
              - function: "IpTables"
                processing_order: 1
              - function: "XFRM"
                processing_order: 2
```

```
Allowed Values:
  • IpTables
  • PF-OpenBSD-PacketFilter
  • Squid
  • StrongSwan
  • XFRM
Source: template_schema.json
```

Figure 7.1: Allowed values for NSFs

- The second scenario, in Figure 7.2 validates the `relationship` property for a node type's `requirements`. Based on the `types_schema.json`, only the established *TOSCA relationships* are allowed. The editor guarantees compliance by providing possible options and reporting any deviations in real-time.

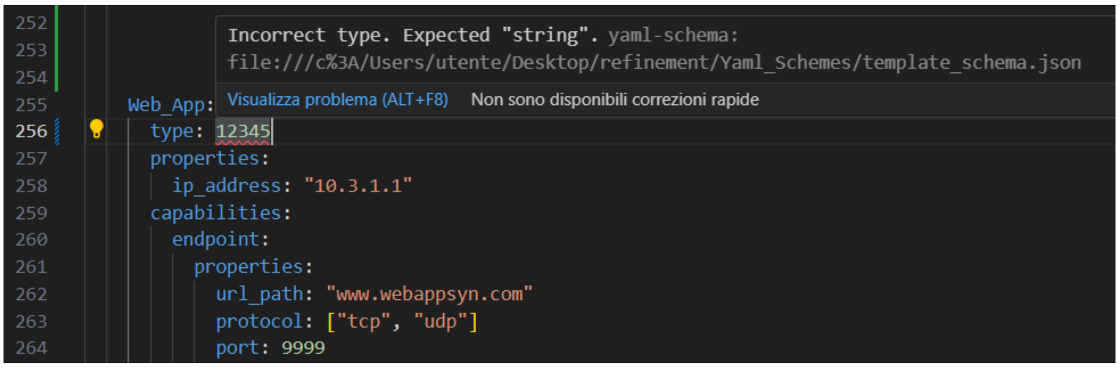


```
node_types:
  company.nodes.NetworkUser:
    capabilities:
      endpoint:
        type: company.
      NSFs:
        type: company.
    requirements:
    - to_subnet:
        capability: to
        node: company.
        relationship: tosca.relationships.network.LinksTo
        occurrences: [1, UNBOUNDED]   #E' necessaria una connessione
```

```
Allowed Values:
  • tosca.relationships.DependsOn
  • tosca.relationships.network.LinksTo
  • tosca.relationships.ConnectsTo
Source: types_schema.json
```

Figure 7.2: Allowed relationships for TOSCA node types

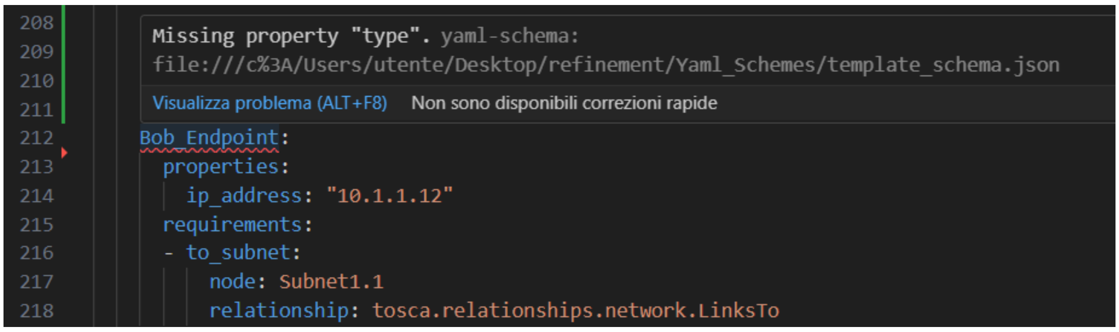
- The Figure 7.3 shows a type mismatch error, where the `type` field under the `Web_App` node is assigned an invalid value (e.g. a *number* instead of a *string*). The validation system, referencing `template_schema.json`, offers immediate feedback indicating the expected type, helping prevent runtime issues.



```
Incorrect type. Expected "string". yaml-schema:
file:///c%3A/Users/utente/Desktop/refinement/Yaml_Schemes/template_schema.json
Visualizza problema (ALT+F8)   Non sono disponibili correzioni rapide
    Web_App:
      type: 12345
      properties:
        ip_address: "10.3.1.1"
      capabilities:
        endpoint:
          properties:
            url_path: "www.webappsyn.com"
            protocol: ["tcp", "udp"]
            port: 9999
```

Figure 7.3: Incorrect data type for a property

- In the case of Figure 7.4, the `type` property is missing in the definition of the `Bob_Endpoint` node. This omission is flagged as an error since the JSON schema mandates the presence of the `type` field for every node. The editor stresses this omission and references the specific schema file where the rule is defined.



```
Missing property "type". yaml-schema:
file:///c%3A/Users/utente/Desktop/refinement/Yaml_Schemes/template_schema.json
Visualizza problema (ALT+F8)   Non sono disponibili correzioni rapide
Bob_Endpoint:
  properties:
    ip_address: "10.1.1.12"
  requirements:
  - to_subnet:
      node: Subnet1.1
      relationship: tosca.relationships.network.LinksTo
```

Figure 7.4: Missing required property in node definition

7.1.2 Dynamically Created Data Classes

The TOSCA YAML parsing process leverages the `TOSCA_types_parser.py` script to dynamically generate *Python* `dataclasses`, representing the structure and properties of customized *TOSCA* `node_types`. A small execution block was added in the script to also enable users to locally run the parser and print out the resulting `dataclasses` with their generated fields and types. These will be used as typed references for instantiating actual elements in the network.

For instance, the dataclass for `company.nodes.CustomSubnet` highlights its fields, such as `cidr` for IP ranges, `negated` for marking those IPs as prohibited, and optional attributes like `ip_version` and `dhcp_enabled` that are derived from the base node type too. Similarly, the `company.nodes.WebApp` dataclass incorporates properties related to supported capabilities, e.g. `endpoint_protocol`, `endpoint_port`, and `endpoint_url_path` from `endpoint` capability. Another notable example is the `company.nodes.MaliciousUser` dataclass, which extends the `company.nodes.CustomSubnet` type by inserting further identifiers, suited to malicious traffic analysis. Lastly, the `company.nodes.NetworkDevice` dataclass emphasizes attributes such as `id` and `NSFs_supported_functions` from `NSFs` capability.

Here are the generated outputs for the examples described above (Listing 7.1):

```
Dataclass for company.nodes.CustomSubnet: company.nodes.CustomSubnet
  negated: typing.Optional[bool]
  cidr: typing.Optional[ipv4_with_negation.IPv4NetworkWithNegation]
  ip_version: typing.Optional[int]
  dhcp_enabled: typing.Optional[bool]

Dataclass for company.nodes.WebApp: company.nodes.WebApp
  ip_address: typing.Optional[ipv4_with_negation
                           .IPv4AddressWithNegation]
  order: typing.Optional[int]
  is_default: typing.Optional[bool]
  endpoint_protocol: typing.Optional[typing.List]
  endpoint_destination_type: typing.Optional[str]
  endpoint_port: typing.Optional[int]
  endpoint_url_path: typing.Optional[str]
  NSFs_supported_functions: typing.Optional[typing.List]

Dataclass for company.nodes.MaliciousUser: company.nodes.MaliciousUser
  mac_address: typing.Optional[str]
  WID: typing.Optional[str]
  DID: typing.Optional[str]
  negated: typing.Optional[bool]
  cidr: typing.Optional[ipv4_with_negation.IPv4NetworkWithNegation]
  ip_version: typing.Optional[int]
  dhcp_enabled: typing.Optional[bool]

Dataclass for company.nodes.NetworkDevice: company.nodes.NetworkDevice
  id: typing.Optional[str]
  NSFs_supported_functions: typing.Optional[typing.List]
```

Listing 7.1: Examples of dynamically generated dataclasses

These dataclasses not only capture the properties and capabilities defined in the TOSCA types, but also integrate advanced data handling through custom types like `IPv4NetworkWithNegation` and `IPv4AddressWithNegation`.

7.1.3 Categorized Dictionaries and Network Graph Representation

The second parsing script, `TOSCA_layout_parser.py`, is responsible for processing *TOSCA* `node_templates` and producing dictionaries and a network graph that together will represent the network layout. These outputs play a central role in the refinement process by providing a concrete view of the *subnets*, *devices*, *entities*, and their *connections*, facilitating security configurations. Below, the key outputs are described along with examples that illustrate their structure and utility:

- The `subnets` dictionary represents the subnets defined in the network topology, including their `CIDR` specification through `IPv4NetworkWithNegation`. Each subnet is uniquely identified and supports the advanced construct *negation* '~' for excluded IP ranges (Listing 7.2).

```
Filtered subnets: {'Subnet1.1': {'cidr': IPv4NetworkWithNegation
    ('10.1.1.0/24')}, ..., 'Internet': {'cidr':
    IPv4NetworkWithNegation('~10.0.0.0/8')}}
```

Listing 7.2: `subnets` dictionary segments

- The `devices` dictionary captures network devices along with their `NSFs_supported _functions` and their `processing_order`. These devices act as enforcement points for security policies (Listing 7.3).

```
Filtered devices: {'Firewall1': {'id': 'firewall-1', '
    NSFs_supported_functions': [{'function': 'IpTables', '
    processing_order': 1}, {'function': 'XFRM', 'processing_order':
    2}]}, ..., 'VPNGateway': {'id': 'vpn-gateway', '
    NSFs_supported_functions': [{'function': 'XFRM', '
    processing_order': 1}, {'function': 'StrongSwan', '
    processing_order': 2}]}}
```

Listing 7.3: `devices` dictionary segments

- The `entities` dictionary enumerates other nodes within the network including, for instance, their *IP* and *MAC addresses*, supported *protocols* and *ports*, plus other additional properties. The example gives an overview of the scope of information captured (Listing 7.4):

```
Filtered entities: {..., 'Bob_Endpoint': {'ip_address':
    IPv4AddressWithNegation('10.1.1.12')}, 'Malicious_UserFULL': {'
    mac_address': '123456789abcdef0123456789abcdef01234', 'cidr':
    IPv4NetworkWithNegation('192.168.0.0/30')}, ..., 'Web_App': {'
    ip_address': IPv4AddressWithNegation('10.3.1.1'), '
    endpoint_protocol': ['tcp', 'udp'], 'endpoint_port': 9999, '
    endpoint_url_path': 'www.webappsyn.com', '
    NSFs_supported_functions': [{'function': 'PF-OpenBSD-
    PacketFilter', 'processing_order': 1}]}, ..., 'VoIP traffic': {'
    endpoint_protocol': ['udp'], 'endpoint_destination_type': 'VoIP
    ', 'endpoint_port': 5060}, ...}
```

Listing 7.4: `entities` dictionary segments

The term "*Filtered*" refers to the removal of empty or unnecessary fields from the dataclass instances for a cleaner output.

Finally, the `connections` object presents the network topology as a *graph*, leveraging the `NetworkX` library. `nodes` represent *subnets*, *devices*, or *entities*, while `edges` symbolize their `relationships`. The graph output is summarized as (Listing 7.5):

```
connections: Graph with 23 nodes and 24 edges
```

Listing 7.5: `connections` output

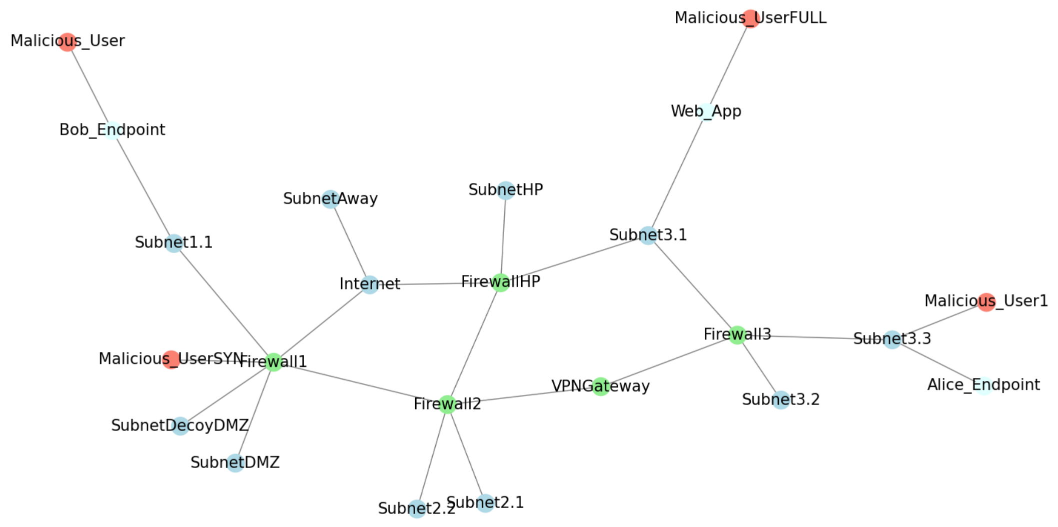


Figure 7.5: NetworkX graph visualization

In addition to the provided data structures, the `networkx_GRAPH.png` reports the *network graph visualization* in Figure 7.5, demonstrating the comprehensiveness of the parsed TOSCA model and displaying the connectivity between network components.

7.1.4 Simulating TOSCA Compatibility in Kubernetes

To verify the compatibility of the *TOSCA YAML model* within the *Kubernetes environment*, the local simulation tool *Kind* was utilized. It enables the creation of lightweight Kubernetes clusters for testing and development, making it suitable for validating this integration.

A simple script was used to parse the defined *TOSCA node types* in `custom_types.yaml` and generate *Kubernetes CRDs* (Custom Resource Definitions), specifically for *CustomSubnet* and *NetworkDevice*. These CRDs were applied to the Kubernetes cluster using the commands (Listing 7.6):

```
> kubectl apply -f customsubnet_crd.yaml
> kubectl apply -f networkdevice_crd.yaml
```

Listing 7.6: CRDs application to Kubernetes cluster

The successful integration of these CRDs was verified through (Listing 7.7):

```
> kubectl get crds
NAME                          CREATED AT
customsubnets.example.com     2024-10-29T16:09:31Z
networkdevices.example.com    2024-10-29T16:09:40Z
```

Listing 7.7: CRDs verification

Using the *TOSCA node templates* in `custom_templates.yaml`, instances of `customsubnet` and `networkdevice` were created. Each instance was inspected to ensure correctness, as demonstrated below (Listing 7.8, Listing 7.9):

```
> kubectl describe customsubnet subnet1.1
Name:         subnet1.1
Namespace:    default
Labels:       app=network-node
              name=subnet1.1
Annotations:  <none>
API Version:  example.com/v1
Kind:         CustomSubnet
Metadata:
  Creation Timestamp:  2024-10-29T16:10:09Z
  Generation:          1
  Resource Version:    13515
  UID:                 7eea89fc-2f54-4ca8-b6c4-216edc1a9401
Spec:
  Cidr:     10.1.1.0/24
  Negated:  false
Events:     <none>
```

Listing 7.8: Example of `customsubnet` CRD instance

```
> kubectl describe networkdevice firewall1
Name:          firewall1
Namespace:     default
Labels:        app=network-node
               name=firewall1
Annotations:   <none>
API Version:   example.com/v1
Kind:          NetworkDevice
Metadata:
  Creation Timestamp:  2024-10-29T16:10:09Z
  Generation:          1
  Resource Version:    13478
  UID:                 c13ddfc8-9f69-459c-a9e4-694b8068eeb1
Spec:
  Id:  firewall-1
  Supported Functions:
    Function:          IpTables
    Processing Order:  1
    Function:          XFRM
    Processing Order:  2
Events:                <none>
```

Listing 7.9: Example of `networkdevice` CRD instance

To simulate *relationships* between TOSCA nodes, the `requirements` specified in the TOSCA model could be mapped to *Kubernetes Service objects*, using `Selector` to reference elements through specific `Labels`. The example output displays how a `LinksTo` relationship between a firewall and a subnet can be handled (Listing 7.10):

```
> kubectl describe service firewall1-linksto-subnet1-1-service
Name:                 firewall1-linksto-subnet1-1-service
Namespace:            default
Labels:               <none>
Annotations:          <none>
Selector:             app=network-node,name=subnet1.1
Type:                 ClusterIP
IP Family Policy:     SingleStack
IP Families:          IPv4
IP:                   10.96.152.189
IPs:                  10.96.152.189
Port:                 <unset>  80/TCP
TargetPort:           80/TCP
Endpoints:            <none>
Session Affinity:     None
Events:               <none>
```

Listing 7.10: Example of K8s Service mapping TOSCA relationship

These tests evidence the potential of CRDs to serve as descriptive representations of TOSCA nodes in Kubernetes. By leveraging CRDs, TOSCA entities can be mapped to Kubernetes resources, to achieve a working network. Some counterpart examples are:

- *Deployments* for network devices (e.g. firewalls and VPN gateways);
- *Services* or *Network Policies* for subnets;
- *Nodes* for other network entities (e.g. endpoints).

This will guarantee the alignment of TOSCA-based network designs with Kubernetes infrastructure, enabling effective deployment and management.

7.2 Extended NSFs Selection Validation

The upgraded mechanism for the selection of suitable *NSFs* was tested with a complex set of *required capabilities* that cannot be satisfied by a single NSF. The new strategy identified and returned *combinations of NSFs* capable of collectively covering all the requested capabilities. The shown outcomes highlight the robustness of the extended approach in addressing potential scenarios where a single NSF is insufficient.

The following *capabilities* were used for the test (Listing 7.11):

```
REQUIRED CAPABILITIES: ['AppendRuleActionCapability', '
    MatchActionCapability', 'TimeStopConditionCapability', '
    SourceAuthActionCapability', '
    IpDestinationAddressConditionCapability', '
    IpSourceAddressConditionCapability', 'AcceptActionCapability', '
    MarginPacketsConditionCapability', 'TimeStartConditionCapability']
```

Listing 7.11: Required capabilities used for the test

7.2.1 NSF Combinations and NSF-to-capabilities Dictionary

The mechanism generated the *combinations of NSFs* below that, together, are capable of satisfying all the required capabilities. They are ranked based on the minimum number of NSFs needed (Listing 7.12):

```
Combinations of NSFs that satisfy all capabilities:
['IpTables', 'StrongSwan']
['IpTables', 'StrongSwan', 'Squid']
['IpTables', 'genericPacketFilter', 'StrongSwan']
['sonaeOperatorOutput', 'StrongSwan', 'IpTables']
['IpTables', 'PF', 'StrongSwan']
['IpTables', 'XFRM', 'StrongSwan']
```

Listing 7.12: Generated combinations of NSFs

This result emphasizes the new functionality to find combinations that minimize the number of NSFs, prioritizing simpler configurations while still ensuring all requested capabilities are met.

A *NSF dictionary*, mapping each NSF to the subset of capabilities it supports among the required ones, is provided as further output (Listing 7.13):

```
NSF dictionary with supported capabilities:
IpTables: ['AppendRuleActionCapability', 'MatchActionCapability', '
    TimeStopConditionCapability', '
    IpDestinationAddressConditionCapability', '
    IpSourceAddressConditionCapability', 'AcceptActionCapability', '
    TimeStartConditionCapability']
StrongSwan: ['SourceAuthActionCapability', '
    IpSourceAddressConditionCapability', '
    IpDestinationAddressConditionCapability', '
    MarginPacketsConditionCapability']
Squid: ['IpSourceAddressConditionCapability', '
    IpDestinationAddressConditionCapability', 'AcceptActionCapability']
genericPacketFilter: ['IpSourceAddressConditionCapability', '
    IpDestinationAddressConditionCapability', 'AcceptActionCapability']
sonaeOperatorOutput: ['IpSourceAddressConditionCapability', '
    IpDestinationAddressConditionCapability', 'AcceptActionCapability']
PF: ['IpSourceAddressConditionCapability', '
    IpDestinationAddressConditionCapability', 'AcceptActionCapability']
XFRM: ['IpSourceAddressConditionCapability', '
    IpDestinationAddressConditionCapability', 'AcceptActionCapability']
ethereumWebAppAuthz: ['AcceptActionCapability']
```

Listing 7.13: NSF-to-capabilities dictionary

This dictionary reports detailed insights, enhancing both the transparency and interpretability of the solutions, and offering a clear understanding of how each NSF contributes to fulfil the capabilities. In this case, `IpTables` emerges as a key NSF due to its wide support for many of the required capabilities, while `StrongSwan` complements it by providing security capabilities like `SourceAuthActionCapability` and `MarginPacketsConditionCapability`.

Concerning this, proceeding with the produced output the system will confirm the selection with the message (Listing 7.14):

```
***NOTIFICATION***: SUITABLE NSFS IpTables, StrongSwan
```

Listing 7.14: Selected combination of NSFs

This outlines how the mechanism selects the most efficient combination available *by default*, prioritizing the one with the fewest NSFs needed to satisfy the set of capabilities. Here, the combination of `IpTables` and `StrongSwan` is chosen as it is at the top of the list of valid solutions.

7.3 Knowledge Base Reprocessing Testing

This section focuses on verifying how the *enhanced refinement process* acts by testing specific scenarios involving *updates to HSPLs*. Differences between an initial set of policies and their subsequent refresh were analyzed, illustrating various cases like *unchanged* policies, modifications with *minor* or *major changes*, *newly* introduced policies, and *removed* ones. The results of these tests, provided as system outputs, will demonstrate the framework's ability to recycle or modify existing configurations and rules, only generating new ones when necessary. By doing so, the approach minimizes computational overhead while maintaining consistency and correctness.

7.3.1 Comparing Initial and Updated Policies

The differences between the initial and updated HSPLs clarify the types of changes the refinement process must handle. Here is a detailed summary of the observed variations (Listing 7.15, Listing 7.16):

- `hspl1`: *existing* with *minor changes*

  The policy retains the same *action* and *object* (`is authorized to access` and `Internet`, respectively), but introduces updates in its *subject* and *optional field*. The *subject* changes from `Alice_Endpoint` to `Subnet3.2`, while the `time period` specified in the *optional field* is updated from `19:00 20:00` to `20:00 21:00`. It is important to note that `Subnet3.2` was already present among the entities in the previous knowledge base, due to its inclusion in the `hspl2` policy.

- `hspl2`: *unchanged*

  This policy remains identical between the initial and updated sets, requiring no further processing during refinement.

- `hspl3`: *removed*

  This policy, initially present, is absent in the updated HSPLs. It will be categorized as "*removed*" during the refinement process.

- `hspl4`: *new policy* with *no match*

  A newly introduced policy with the *action* `protect confidentiality`, which has not appeared in the last set. This absence of any match with previous HSPLs necessitates full rule generation from scratch.

- `hspl5`: *new policy* with *major changes*

  While this policy is new, it shares similarities with `hspl2` in its *action* (`is not authorized to access`) and *object* (`Web_App`). However, it replaces the *subject* `Subnet3.2` with `Subnet1.1`. It must be considered that `Subnet1.1` is not present among the entities recorded in the previous knowledge base, leading to its categorization as a new policy attempting to be addressed through partial reconfiguration.

```
<hspl id="hspl1">
    <subject>Alice_Endpoint</subject>
    <action>is authorized to access</action>
    <object>Internet</object>
    <optionalField>
        <optionType>time period</optionType>
        <optionValue>19:00 20:00</optionValue>
    </optionalField>
</hspl>

<hspl id="hspl2">
    <subject>Subnet3.2</subject>
    <action>is not authorized to access</action>
    <object>Web_App</object>
</hspl>

<hspl id="hspl3">
    <subject>Alice_Endpoint</subject>
    <action>protect integrity</action>
    <object>Bob_Endpoint</object>
</hspl>

```

Listing 7.15: Initial policies

```
<hspl id="hspl1">
    <subject>Subnet3.2</subject>
    <action>is authorized to access</action>
    <object>Internet</object>
    <optionalField>
        <optionType>time period</optionType>
        <optionValue>20:00 21:00</optionValue>
    </optionalField>
</hspl>

<hspl id="hspl2">
    <subject>Subnet3.2</subject>
    <action>is not authorized to access</action>
    <object>Web_App</object>
</hspl>

<hspl id="hspl4">
    <subject>Alice_Endpoint</subject>
    <action>protect confidentiality</action>
    <object>Bob_Endpoint</object>
</hspl>

<hspl id="hspl5">
    <subject>Subnet1.1</subject>
    <action>is not authorized to access</action>
    <object>Web_App</object>
</hspl>

```

Listing 7.16: Updated policies

7.3.2 Analysis of Refinement Outputs

The outputs from the refinement process reveal how the system categorizes and processes the updated HSPLs based on their differences from the initial set (Listing 7.17). Below, each HSPL state and the corresponding outputs logged in `refinement.log` are examined:

```
HSPLS STATES: {'new': {'no_match': ['hspl4'], 'minor_changes': [],
'major_changes': [('hspl5', 'hspl2')]}, 'existing': {'minor_changes':
['hspl1'], 'major_changes': [], 'unchanged': ['hspl2']}, 'removed': {
'removed': ['hspl3']}}
```

Listing 7.17: HSPLs categorization

New HSPLs

- **hspl4**

```
State new:
    Change Type: no_match
        - HSPL ID: hspl4
```

Listing 7.18: `hspl4` state

The `hspl4` policy, introducing the new *action* `protect confidentiality`, finds "*no match*" in the previous HSPLs due to the action's uniqueness (Listing 7.18). As a result, a new configuration is required, as indicated by the following log entries (Listing 7.19):

```
FIND CONF 10.3.3.24 10.1.1.12 hspl4
source-dest 10.3.3.24 10.1.1.12
Subnet3.3 Subnet1.1
...
***NOTIFICATION***: SUITABLE NSFS XFRM
...
SELECTED DEVICE(s) TO CONFIGURE: {'Firewall1'}
```

Listing 7.19: `hspl4` configuration

The configuration determines that `Firewall1` is suitable for the task, with the `XFRM` NSF selected to handle the required capabilities. Furthermore, new rules are generated from scratch (Listing 7.20):

```
Requested Capabilities:
 [
  {
    "capability": "EncryptionActionCapability",
    "detail": "",
    "hsplid": "hspl4"
  },
  ...
]

Mode: ['CONFIDENTIALITY']

Generated Rules:
 [
  {
    "hsplid": "hspl4",
    "device": "Firewall1",
    "nsf": "XFRM",
    "capabilities": [ ... ]
  },
  ...
]
```

Listing 7.20: `hspl4` rules generation

- **hspl5**

```
State new:
    Change Type: major_changes
        - HSPL ID: ('hspl5', 'hspl2')
```

Listing 7.21: `hspl5` state

The `hspl5` policy finds a partial match with `hspl2`, sharing the same *action* and *object* (Listing 7.21). Although, its different *subject*, `Subnet1.1`, does not appear in the previous knowledge base, necessitating configuration (Listing 7.22):

```
FIND CONF 10.1.1.0/24 10.3.1.1 hspl5
source-dest 10.1.1.0/24 10.3.1.1
Subnet1.1 Subnet3.1
...
***NOTIFICATION***: SUITABLE NSFS IpTables
...
SELECTED DEVICE(s) TO CONFIGURE: {'Firewall1'}
```

Listing 7.22: `hspl5` configuration

Unlike `hspl4`, `hspl5` does not require rule generation again. Instead, rules from `hspl2` are recycled, with necessary modifications to the details of capabilities like `IpSourceAddressConditionCapability` and `IpDestinationAddressConditionCapability` to adapt them to `hspl5`, respectively to the different rules because of the policy's bidirectionality. The logs reflect this (Listing 7.23):

```
- Modification detected in capability:
    IpSourceAddressConditionCapability
- Configured device Firewall3 replaced with Firewall1 for HSPL hspl5
- Rule recycled for HSPL hspl5 on Firewall1 with IpTables
- Modification detected in capability:
    IpDestinationAddressConditionCapability
- Configured device Firewall3 replaced with Firewall1 for HSPL hspl5
- Rule recycled for HSPL hspl5 on Firewall1 with IpTables
```

Listing 7.23: `hspl5` logs

Additionally, the device for configuration changes from `Firewall3` (used for `hspl2`) to `Firewall1` due to updates in the intermediate paths, but it is sufficient to replace the latter in the rules, as the device's configuration remains the same.

Existing HSPLs

- **hspl1**

```
State existing:
    Change Type: minor_changes
        - HSPL ID: hspl1
```

Listing 7.24: `hspl1` state

Even though `hspl1` changes its *subject* and *optional field*, the *subject* `Subnet3.2` already occurs in the knowledge base as a consequence of `hspl2`, and the intermediate paths remain unchanged, as can be verified on the previous network graph in Figure 7.5. Thus, no new configuration or rule generation is needed (Listing 7.24). That is enough to update the existing rules, as seen in the logs (Listing 7.25):

```
- Modification detected in capability:
    IpSourceAddressConditionCapability
- Modification detected in capability: TimeStartConditionCapability
- Modification detected in capability: TimeStopConditionCapability
- Rule updated for HSPL hspl1 on Firewall3 with IpTables
- Rule updated for HSPL hspl1 on Firewall3 with IpTables
```

Listing 7.25: `hspl1` logs

The modifications adjust the capabilities related to the *source IP address*, which passes from a *single IP* (`Alice_Endpoint`) to a *CIDR range* (`Subnet3.2`), and to the `time period` as expected by the new *optional field*.

- **hspl2**

```
State existing:
    Change Type: unchanged
        - HSPL ID: hspl2
```

Listing 7.26: `hspl2` state

The `hspl2` policy is maintained unaltered in both sets, recording the relative configurations and intermediate rules in the files as they were (Listing 7.26).

Removed HSPLs

- **hspl3**

```
State removed:
    Change Type: removed
        - HSPL ID: hspl3
```

Listing 7.27: `hspl3` state

The `hspl3` policy is absent from the novel set. Consequently, all associated data is deleted from the renewed knowledge base and intermediate rules (Listing 7.27).

Chapter 8

Conclusions and Future Works

This thesis focuses on enhancing the *automated refinement* of *HSPLs* into enforceable configurations suitable for diverse and articulated network environments. Building upon prior frameworks that have seen the participation of several interested parties, both researchers and students, it addresses essential challenges in scalability and adaptability, introducing solutions to extend the functionalities of the refinement process.

The starting point was a framework that effectively transformed abstract HSPLs into configurations executable by network security devices. However, it revealed certain limitations, particularly in handling dynamic network architectures and evolving security requirements. As modern systems expect greater flexibility and precision, this research aimed to meet the identified needs by proposing aligned advancements.

Key contributions of this thesis include the integration of a standard for describing network layouts, a method to identify comprehensive combinations of *NSFs* to satisfy diversified security conditions, and a refined strategy for reprocessing knowledge bases. These innovations collectively elevate the tool's expertise in managing network policies, optimizing configuration updates, and improving the overall refinement workflow.

The adoption of the *TOSCA YAML model* addressed the lack of a structured manner for defining network layouts. This integration provided a clear representation of network entities and their interconnections, also being compliant with cloud-native environments like *Kubernetes*. By enabling the definition and validation of network components and their relationships, the approach safeguards consistency and reduces manual errors.

The extension of the *NSF-Catalogue querying mechanism* allowed for the identification of combinations of *NSFs* to fulfill heterogeneous sets of security capabilities. This feature significantly enhances the versatility of the tool by accommodating scenarios where no single NSF is sufficient. Furthermore, it ensures that the refinement process remains practical by prioritizing efficient combinations, namely those that minimize the number of necessary capabilities.

The *knowledge base strategy* was designed to introduce reprocessing of past intermediate information, optimizing the management of configuration updates in response to policy changes. By distinguishing between *new*, *modified*, or *removed* policies, the tool avoids redundant operations, applies proper adjustment, and adapts existing rules to align with transforming specifications. This minimizes computational overhead, while securing the system reflects intended policy objectives.

The implementation phase concentrated on embedding the proposed achievements into the existing refinement framework. Each step was developed employing adequate techniques, such as structured parsing and consequent dataclass generation for the network layout model, splitting requested capabilities across multiple NSFs, and knowledge base driven workflows. These solutions delivered a solid foundation for stable and reliable automated security policy refinement.

The validation of results confirmed the relevance of the reached progress through targeted testing across the three main covered areas. In particular, it demonstrated the ability of the framework to interpret detailed network data descriptions, generating related configurations and supporting policy updates even under complex scenarios. Therefore, it was verified the fulfillment of the thesis purposes, overcoming the deficiencies outlined at the beginning of this research.

8.1 Future Works

Future research opportunities arising from this thesis can call attention to expanding the scope and applicability of the presented advancements. One core direction involves further exploration of the *TOSCA YAML model*. This includes utilizing additional TOSCA standard resources to represent a broader range of network components and deepen *Kubernetes* integration through simulations and operational network scenarios, enabling dynamic scaling and fault tolerance.

Another potential field is providing the *NSF-Catalogue* with an API to supply NSF combinations directly, streamlining automation and interfacing with third-party systems.

One more direction for upcoming studies concerns introducing validation mechanisms during the knowledge base reprocessing, specifically the integration of automated checks for redundancies, conflicts, or anomalies when handling minor or major modifications, as well as policy removals. Through these controls, the framework could ensure avoiding overlapping or contradictory rules.

Higher-level developments could focus on incorporating real-time monitoring to constantly update the descriptive network model, paired with adaptive refinement strategies to address dynamic conditions and emerging threats. Additionally, integrating machine learning techniques offers opportunities for predictive policy refinement, proactive threat detection, and optimization of NSFs selection.

Bibliography

[1] Fulvio Valenza, Antonio Lioy, et al. «User-oriented Network Security Policy Specification.» In: *J. Internet Serv. Inf. Secur.* 8.2 (2018), pp. 33–47. URL: https://iris.polito.it/retrieve/e384c430-ba6d-d4b2-e053-9f05fe0a1d67/jisis-2018-vol8-no2-03.pdf.

[2] D.C. Verma. «Simplifying network administration using policy-based management». In: *IEEE Network* 16.2 (2002), pp. 20–26. DOI: 10.1109/65.993219. URL: https://ieeexplore.ieee.org/abstract/document/993219.

[3] Xia Yang and Jim Alves-Foss. «Security Policy Refinement: High-Level Specification to Low-Level Implementation». In: *2013 International Conference on Social Computing*. 2013, pp. 502–511. DOI: 10.1109/SocialCom.2013.77. URL: https://ieeexplore.ieee.org/document/6693374.

[4] Joseph C Giarratano et al. «CLIPS User's guide». In: *NASA Technical Report, Lyndon B Johnson Center* (1993). URL: https://clipsrules.net/documentation/v631/ug631.pdf.

[5] Charles L. Forgy. «Rete: A fast algorithm for the many pattern/many object pattern match problem». In: *Artificial Intelligence* 19.1 (1982), pp. 17–37. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(82)90020-0. URL: https://www.sciencedirect.com/science/article/pii/0004370282900200.

[6] Mattia Bencivenga. «Towards the automatic refinement of high-level security policies in computer networks». MA thesis. Politecnico di Torino, 2022. URL: https://webthesis.biblio.polito.it/22803/.

[7] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. «Yaml ain't markup language (yaml™) version 1.1». In: *Working Draft 2008* 5.11 (2009). URL: https://yaml.org/spec/history/2004-12-28/2004-12-28.pdf.

[8] Christoph Kleine. «Backward and forward compatibility for TOSCA simple profile in YAML version 1.0: concept and modelling tooling support». MA thesis. 2017. URL: https://www2.informatik.uni-stuttgart.de/bibliothek/ftp/medoc_restrict.ustuttgart_fi/MSTR-2017-50/MSTR-2017-50.pdf.

[9] Woramon Chareonsuk and Wiwat Vatanawood. «Translating TOSCA Model to Kubernetes Objects». In: *2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. 2021, pp. 311–314. DOI: 10.1109/ECTI-CON51831.2021.9454890. URL: https://ieeexplore.ieee.org/abstract/document/9454890.

[10] Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoč. «JSON: Data model and query languages». In: *Information Systems* 89 (2020), p. 101478. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2019.101478. URL: https://www.sciencedirect.com/science/article/abs/pii/S0306437919305307.

[11] 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)*. 2022, pp. 474–479. DOI: 10.1109/NetSoft54395.2022.9844057. URL: https://ieeexplore.ieee.org/abstract/document/9844057.

[12] Aurelio Cirella. «An abstract model of NSF capabilities for the automated security management in Software Networks». MA thesis. Politecnico di Torino, 2022. URL: https://webthesis.biblio.polito.it/secure/22805/1/tesi.pdf.

[13] Hector J Levesque and Gerhard Lakemeyer. *The logic of knowledge bases*. Mit Press, 2001. URL: https://books.google.it/books?hl=en&lr=&id=4MzyIJibf-UC&oi=fnd&pg=PR15&dq=logic+of+knowledge+bases&ots=4-uQRKBT_B&sig=WQ7d-fLlsgOo3a93hNxNiKBX2sk&redir_esc=y#v=onepage&q=logic%20of%20knowledge%20bases&f=false.

[14] STS POLITO and Manjon Caliz. «D4. 4 Security and Certification IT2 integration». In: (2023). URL: https://fishy-project.eu/sites/fishy/files/public/content-files/deliverables/D4.4%20Security%20and%20Certification%20IT2%20integration_v1.0.pdf.

[15] Cataldo Basile, Gabriele Gatti, and Francesco Settanni. «A Formal Model of Security Controls' Capabilities and Its Applications to Policy Refinement and Incident Management». In: *arXiv preprint arXiv:2405.03544* (2024). URL: https://arxiv.org/pdf/2405.03544.