POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

Correctness Validation and Scalability Analysis of Stateful Firewalls in the VEREFOO Framework

Supervisors
Prof. Fulvio Valenza
Prof. Riccardo Sisto
Prof. Daniele Bringhenti

Candidate
Gabriele Iannace

Academic Year 2025-2026

Abstract

Network models like Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) have greatly impacted the manner in which we architect and manage our networks today. These models allow for control over data flow with the help of central control. This is done through the decoupling of the data processing aspect and the decision-making aspect. These models also support a networking construct wherein the networking functions can be developed on ordinary servers. This results in a network environment that is dynamic and vibrant to cater to the demands of current services. The increasing demands of security management, especially in terms of configuring Network Security Functions (NSFs) like packet filters or firewalls, is becoming increasingly difficult and demanding in today's ever-changing and dynamic environments. Previously, security configurations and management were done manually. These operations took a considerable amount of time to accomplish. This resulted in ineffective & sluggish responses to meet the demands at a rapid pace. At the same time, the manual nature of the task resulted in errors occurring regularly.

The novel contribution of this research is the VEREFOO (Verified Refinement and Optimized Orchestration) framework to tackle the urgent security management, applicable to heterogeneous virtual networks within quickly evolving settings. VEREFOO is a security automation framework designed to provide effective, efficient orchestration of automated deployment and configuration of virtual network firewalls through a verified, heuristic approach. The MaxSMT problem of automatic deployment and configuration of firewalls in virtual networks is formulated and solved with the z3 theorem prover to reduce the number of deployed firewalls and implemented filtering rules under the initial condition that everything is satisfiable.

This thesis will concentrate on both proving the logical correctness of VEREFOO and evaluating its scalable performance; specifically in relation to the integration of stateful firewalls that filter bidirectional flow. To achieve the above goal, we needed to prove the effectiveness of the logical formulation that models the stateful semantics of stateful firewalls and the ALLOW COND (Permit Conditionally) action. Chapter 4 evaluates the correctness of VEREFOO through the development of a rigorous cross-validation methodology in order to validate that the configurations produced by VEREFOO refine the NSRs and do so in a manner that does not introduce inconsistencies.

Following the evaluation of correctness, Chapters 5 and 6 assess scalability within complex scenarios by creating a test case that replicates the Texas 2000 model, a cyber-physical network representing a smart electric power grid. Results from this test case demonstrate that the optimization techniques (removal of logical quantifiers and auto-configuring pruning) utilized in the VEREFOO framework allow it to be computationally viable and reliable in large-scale architectures. Thus, VEREFOO offers a viable alternative to manually configuring and maintaining distributed security configurations for complex architectures. The final section of this thesis outlines the results of this study along with recommendations for future research.

Contents

Chapter 1

Stateful firewalls

Stateful Firewall Defense represents one of the main themes in the present work because of its strategic importance in managing traffic flow and security at the network layer. The present Chapter therefore aims at providing a comprehensive review of the theoretical and technological aspects of stateful Firewalls, while also describing the technologies that can be used to configure them in order to manage the security of a network.

This Chapter starts by defining the concept of a "Stateful Function", and its importance in networking environments, which have been significantly affected by the introduction of SDN and NFV paradigms. The Stateful Function represents a System whose operation depends on its being able to maintain an internal state among the several calls it receives; in contrast, Stateless Systems do not store any information from call to call.

Therefore, the present Chapter starts by reviewing the security context in modern networks (Section 1.1.1) to highlight the evolution of threats and the growing necessity of traffic control, in addition to identifying the central role played by firewalls as the primary means of defense against these threats.

Then the present Chapter describes the key differences between Stateful and Stateless Firewalls (Section 1.1.2) and explains the reason why state management is a necessary condition for connection monitoring.

In Section 1.2 and Section 1.3 the authors examine the applicability of Stateful Functions in the context of Networks, by presenting the generic architecture of a Stateful Firewall and by explaining how the latter maintains a State Table (also referred to as Connection Table) containing information regarding all the current connections in the network (the IP addresses and Ports involved, the state of each connection).

A particular focus is given to the ability to identify and process packets in different States (i.e., New, Established, Related, Invalid), and to explain the advantages of using such States in distributed and dynamic Architectures.

Section 1.4 is devoted to the specific mechanisms of State Management and Connection Tracking that represent the functional core of a Stateful Firewall. In this section the authors analyze how the firewall dynamically constructs and updates the State Table, with particular reference to the key role of Inactivity Timers (Timeouts) in eliminating Orphaned Connections automatically, and they discuss the inherent limitations of such approach, including the optimizations that must be performed to prevent the risk of

Saturation of the State Table.

Finally, Section 1.5 closes the present Chapter by illustrating the typical tools and technologies that are used for the configuration of Stateful Firewalls in Real World Scenarios.

Platforms such as Iptables (with special regard to the conntrack Module, which enables State Management), IpFirewall, Open vSwitch are reviewed; these platforms are expected to be useful during the Thesis Implementation Phase for Policy Translation. Therefore, the present Chapter supplies a detailed Conceptual Background to understand the functionalities of Stateful Firewalls, so that Chapter 2 may go into detail about the VEREFOO Framework and the formal methodologies adopted for the Orchestration and Automatic Configuration of such systems in Virtualized Environments, in order to overcome the Limitations of Manual Configuration.

1.1 Introduction

1.1.1 Network Security Environment

The development of new *paradigms of software networking*, like **Software-Defined Networking (SDN)** and **Network Functions Virtualization (NFV)**, has dramatically transformed the landscape of communication infrastructure systems [7]. The purpose of these two technologies is to create **flexibility and speed** in managing networks; they allow network functions (NF) to be detached from hardware and enable the dynamic creation of complex logical networks, called **Service Graphs (SG)** [3].

However, the greater complexity in structure and the increased dynamism in SDN and NFV-based environment have made security management even more difficult. It is now **practically impossible** to correctly configure and timely activate defense tools with manual methods [6]. Traditional security management methods, based on human intervention, are therefore inherently **liable to human errors** (*human errors*) and generate serious *misconfiguration*, representing one of the most common vectors of attacks. Furthermore, manual configuration takes **considerable time** and generates a **relevant reaction delay** with respect to operational changes or the emergence of new threats [7].

Therefore, given the **continual evolution of cyber threats**, the need for **strictly and automatically managed traffic** has become indispensable [14]. For security in today's networks, **network security requirements (NSRs)** must be strictly defined – for example, to separate nodes and/or ensure a specific level of connectivity – so as to avoid exploiting vulnerabilities or misconfigured configurations. To overcome the limits of a manual approach, **security automation** has become an essential theme [3]. The double goal of the approach is to decrease the risk of human error and to decrease reaction times, while at the same time creating **correct and optimized solutions formally** [7].

In this context, **firewalls** (or packet filters), continue to play the traditional role of **the first line of defense** and remain the **Network Security Function (NSF)** and the **most commonly employed defense mechanism** [6]. As opposed to traditional contexts, characterized by a single point of control, modern SDN/NFV architectures promote the use of **distributed architectures**, in which multiple virtual firewall instances are strategically located throughout the *Service Graph*. The configuration and orchestration

of these distributed defenses cannot be assigned to human interventions and require the use of formal methodologies, such as those derived from MaxSMT problems, to guarantee that the assignment and filtering policies are optimal [7]. However, to establish an effective defense in such architectures, it is necessary to analyze the various types of filtering available. The fundamental differentiation, which will be examined further below, is related to how communications are monitored: the difference between firewalls that do not store the results of past communications (stateless) and those that, instead, continuously monitor the status of communications (stateful) [4].

1.1.2 Difference between stateless and stateful firewalls

The most significant difference defined in the last section involves how firewalls maintain a decision-making context.

Stateless Packet Filtering (without State) represents the old way of doing things. Each packet is evaluated and filtered as an individual event, and based solely upon the packet's headers (for example; IP addresses, Ports), the decision will be made (allow/deny) against a predefined static list of rules. Although this provides low latency and minimal processing power, the inability to associate packets together renders it powerless to sophisticated attacks and difficult to configure for protocols that have return flow requirements [14].

In order to address the above mentioned shortcomings, the idea of "State" in Connection Monitoring was established. This provided the basis for a Firewall that is Stateful (based on State). The Stateful Firewalls actively monitor the state of current connections to the network. The heart of the architecture is the State Table, a cache area used by the Firewall to store information regarding each current valid session (for example; IPs, ports, sequence number).

Upon receipt of a packet attempting to establish a new connection (state `NEW`), the packet will be checked against the Policy Rules. If the connection is allowed, an entry will be placed in the State Table. Upon receipt of subsequent packets belonging to that session (states `ESTABLISHED` or `RELATED`) they will be compared to the State Table, and if found, will be allowed to pass through quickly without having to go through the majority of the Rule Analysis [14].

The greatest benefit of the Stateful Approach lies in the ability to provide a more granular level of security and the capability to make informed decisions, and therefore detect abnormal or anomalous packets or attack attempts (such as Session Hijacking), that would be unable to be identified by Stateless Filters. Additionally, the Stateful Approach is able to reduce the computational overhead associated with managing return traffic.

However, the greatest drawback of the Stateful Approach is the increased resource utilization, particularly the Memory required to maintain the State Tables, which can act as a Bottleneck or as a target for Denial-of-Service (DoS) type attacks intended to flood the State Tables.

Below is a summary of the Key Differences between the Two Approaches presented in the form of a Comparative Table.

Characteristic	Stateless Firewall	Stateful Firewall
Filtering Logic	Analyzes each packet individually	Analyzes packets in the context of a session
Context	No context from previous connections	Maintains a "state table" of active connections
Decision	Based on static rules (e.g., IP, port)	Based on rules and the current connection state
Rule Complexity	Simple but rigid, potentially verbose	More sophisticated, reduces the number of explicit rules
Overhead	Low	Higher (memory for state table, CPU for state management)
Security	Limited, vulnerable to attacks that exploit the state (e.g., spoofing, session hijacking)	High, detects out-of-context packets and blocks specific state attacks
Ideal Applications	Low-traffic environments, simple requirements, layer 3/4 DoS mitigation	Complex environments, transactional applications, granular session control

Table 1.1: Comparison between stateless firewalls and stateful firewalls

1.1.3 Reasons for using stateful firewalls in modern architectures

The use of stateful firewalls based on their state in today's network architecture is driven by the highly demanding technical requirements that do not allow stateless filters to be met. These have primarily been caused by an increasingly strong demand for greater control and reliability and the ability to handle complex protocols within distributed environments [14].

Its most important benefit is in providing more detailed and secure traffic management. Stateful firewalls can develop a state table and therefore are able to understand packets as part of a logical communications session, rather than just as separate packets. Thus, they are able to make "informed decisions" (*informed decisions*), which results in a substantially higher level of security as the correlation of packets enables them to recognize and counter advanced attacks that rely on the manipulation of state, e.g. *session hijacking*, *spoofing*, or transmitting anomalous packets (e.g., `INVALID` or *out-of-context*), that would otherwise go unrecognized by a stateless filter [4].

This logic is critical for managing complex protocols (e.g., TCP, UDP, etc.), and in a broader sense, bidirectional traffic flows. For connection-oriented protocols (e.g., TCP), a stateful firewall needs to be able to identify the initial phase of a communications session (i.e., `NEW`), and then track the response packets (`ESTABLISHED` or `RELATED`), which requires a state-based system. In this way, the firewall can quickly permit the passage of return traffic (`ESTABLISHED/RELATED`) by performing a direct lookup against the entry in the state table, rather than having to perform a full analysis of all of the *Filtering Policy* rules for each packet after the first.

Stateful firewalls offer additional benefits related to resource efficiency and optimization,

both of which are essential elements in dynamic and distributed environments [7]. Since a stateful firewall will permit the passage of return traffic (`ESTABLISHED/RELATED`) using a simple and fast *lookup* in the state table, it does not have to perform the complete and computationally intensive analysis of the entire *Filtering Policy* for most of the packets of a session. Additionally, stateful firewalls will improve the *throughput* of the firewall, and will greatly simplify policy definition; it is no longer required to write complex and numerous static rules to describe the flow of return traffic, which reduces the attack surface that arises from *misconfiguration* [3].

1.2 Architecture and operating principles

In establishing the limitations of the stateless method in the preceding sections, as well as the technological factors leading to the selection of a state-based screening process, we can now evaluate the technical and operational characteristics enabling stateful firewalls to function. The primary objective in evaluating how a stateful firewall technically provides for both context management and session memory is to break down the system into its basic logical elements.

The purpose of this section is to evaluate the **operational model** of a stateful firewall (Section 1.2.1), which defines the logical path followed by a packet. Additionally, it will examine the **architectural element: state table management**, with particular focus on the creation, update, and removal of connections (Section 1.2.2). Next, the section will discuss how this model evaluates both **inbound and outbound traffic** flows (Section 1.2.3). Finally, the section will establish the **primary components** making up the structure (Section 1.2.4) of the stateful firewall's architecture, including the *packet filter*, the *connection tracker*, and the *policy engine* [14].

1.2.1 General model of a stateful firewall

The operating model of a stateful firewall can be explained by using a logical design that views network traffic as a series of communication flows rather than individual, independent events. With this view of traffic, static filtering limitations are overcome and a dual decision-making process is implemented in order to differentiate how the first packets of a connection are processed versus subsequent packets within connections that have already been established.

A basic logical architecture (block diagram) is composed of the interaction between two major components: the *Filtering Policy* (static access rules) and the *State Table* (also referred to as a *connection table*), a dynamic component used as a memory for the number of current connections. The interaction between these two components determines the path taken by a packet.

A generic description of the normal operational procedure for the firewall is given below:

1.*Packet Arrival and State Table Search:* When a packet arrives at an interface of the firewall, the firewall performs a *search* of the *State Table* in order to determine whether the packet has a relationship to any of the currently monitored connections. There are

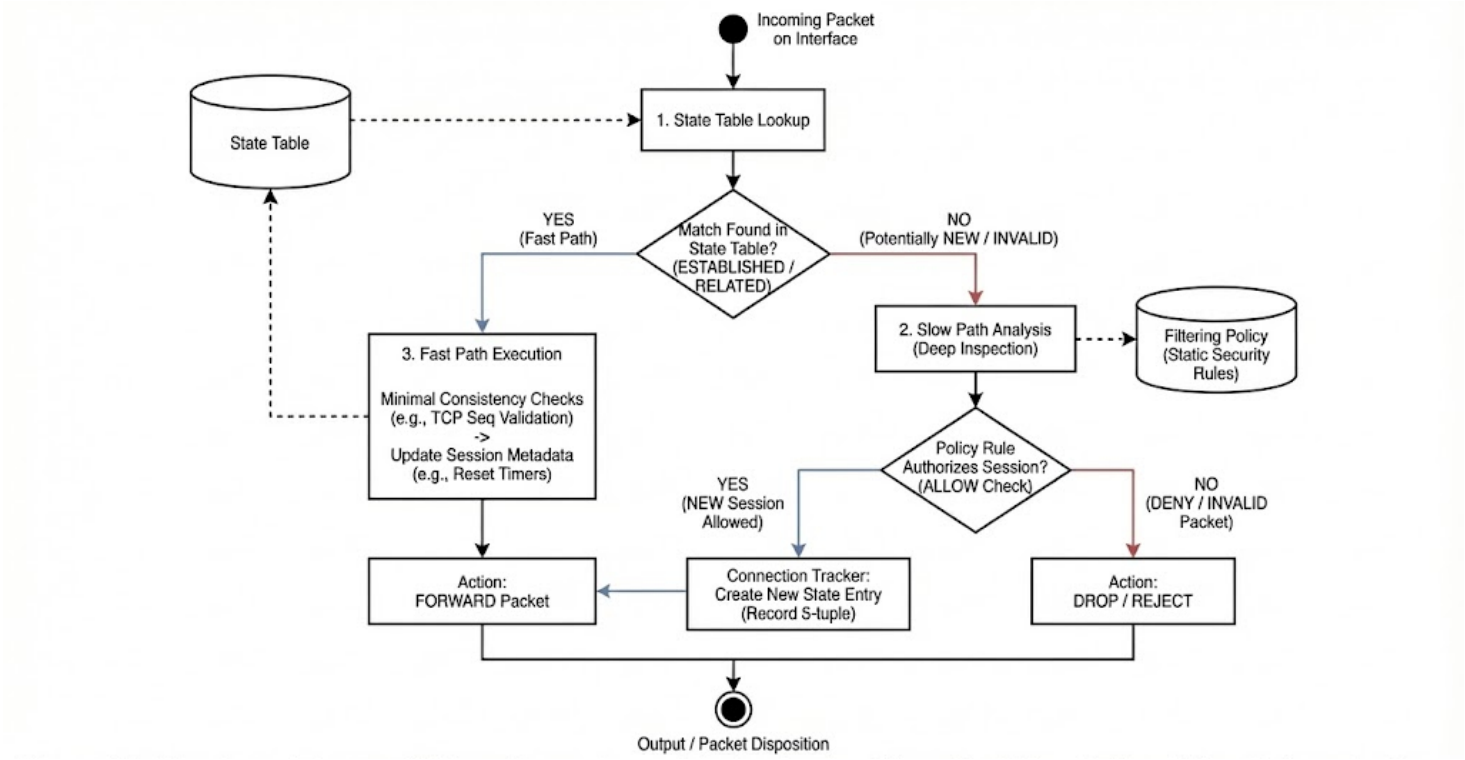


Figure 1.1: Flowchart of the stateful operation model.

two different operational paths for this search: the *slow path* and the *fast path*. These terms are commonly used in literature.

2.*Slow Path (Creation and Management of New Connections):* If the packet being searched is not found in the *State Table*, it is considered to be `NEW`. The packet is then sent to the *main analysis path* or the *slow path*. Within this path, the packet is examined in detail relative to the *Filtering Policy* (all applicable security policies). As a result, the engine must examine the packet against many rules. Each examination is computationally intensive because the engine must compare the packet against each of the applicable rules. If there is a rule contained in the *Policy* that allows the initiation of a session (`ALLOW`), the *Connection Tracker* (*State Inspection Engine*) generates a new entry into the *State Table* with information about the session (commonly includes the five-tuple and its reverse) and sets the session status to `INITIALIZED`. The packet is then passed to the next hop. If none of the rules allow the establishment of the connection, the packet is dropped (`DROP` or `REJECT`) and no record of the session is maintained.

3.*Fast Path (Management of Existing Sessions):* If the incoming packet finds a match in the *State Table* (i.e., it is part of an `ESTABLISHED` or `RELATED` session), it is routed to the *fast path*. For this optimal path, the system bypasses the complete and resource-intensive analysis of all applicable rules in the *Filtering Policy*. The system performs only limited validation (e.g., validates the sequence numbers of the TCP segments to detect *session hijacking*) and updates the metadata of the session in the *State Table* (e.g., resets the

timeout counters) before passing the packet to the next hop [14].

4.*Invalid Packet Management:* If a packet does not correspond to an existing connection and cannot be determined to be a valid packet to initiate a new session (e.g., an ACK with no prior NEW session), it is classified as `INVALID` and normally discarded (`DROP`).

The use of the slow-path (a full and "resource-intensive" inspection for `NEW` packets) and fast-path (a simple and "lightweight" lookup for `ESTABLISHED`/`RELATED` packets) represents the fundamental mechanism used by stateful firewalls to achieve a balance between strict and fine-grained security controls and high throughput for legitimate communications.

The detailed management of the *state table*, including the structure of the table, and the mechanisms for updating entries in the table will be discussed in Section 1.2.2.

1.2.2 State table management (connections and sessions)

The *state table* is the dynamic *high-performance data structure* responsible for enabling the efficiency of the *fast path* of a stateful firewall; its purpose is to act as a memory of context for all active communication sessions going through the firewall. The state table allows for *informed decision-making regarding filtering*, beyond simple packet-by-packet analysis.

The state table is populated and managed by a *critical process* that manages the *lifecycle* of the entries within the table, from creation to deletion. Each entry within the table contains a minimum amount of *information required to uniquely identify an ongoing session*. While there is no strict definition of what should be contained within an entry, generally the following elements are included:

- The *five-tuple* (Source IP Address, Source Port Number, Destination IP Address, Destination Port Number, L4 Protocol), used as the *unique key* to identify the flow;
- The *reverse five-tuple* (Destination IP Address, Destination Port Number, Source IP Address, Source Port Number, L4 Protocol), calculated and stored before the session is closed, to allow for rapid lookup of return traffic.
- The *current protocol state*, either at the application layer or the transport layer (i.e., the current state of a TCP three-way handshake, or states like ESTABLISHED, RELATED);
- An *inactivity timer*, which defines how long a session can remain idle, i.e., without receiving or sending data, until it is terminated due to lack of activity.

In addition to the above elements, other *metadata* could also be included within an entry (such as packet and byte counts, TCP sequence number validation flag).

Each entry in the state table has a *lifecycle* managed by the *Connection Tracker*:

Creation (Create): When a packet is classified as NEW (as defined in Section 1.2.1) and successfully inspected against the *Filtering Policy* in the slow path, a new entry is created. By creating an entry for each new session, the stateful firewall limits the

allocation of state resources to legitimate and authorized sessions, while preventing Denial of Service (DoS) attacks that attempt to flood the state table with unauthorized traffic.

Update (Update/Read): Most of the time spent managing the entries in the state table is associated with updating existing sessions. When a session is detected by the fast path, the firewall rapidly looks up the entry corresponding to the arriving packet, determines whether the received packet matches the record of the session in the table, and if so, updates the entry. The primary operation performed during an update is the reset of the inactivity timer, effectively extending the lifetime of the session. Other metadata, such as internal protocol state (for example, TCP window sizes) can also be updated.

Deletion (Delete): Deletions of entries in the state table are critical to maintain the integrity and availability of the table. There are two ways in which deletions occur:

- **Explicit Closure:** The Connection Tracker deletes the entry after the firewall detects packets indicating an orderly shutdown of the session (for example, FIN/RST flags for TCP).
- **Timeout:** This is an implicit garbage collection mechanism. Entries are deleted when the session does not receive or send data for longer than the time specified by the inactivity timer. This helps prevent orphaned sessions (those that have been interrupted or whose state has become invalid) and their state to accumulate in the table, thus preserving the ability to manage new connections and optimizing memory utilization.

The performance and reliability of a stateful firewall depends on the *robustness* and *optimization* of the state table management mechanisms described. The protocol-specific timeout parameter analysis and optimization strategies for state table management will be examined in detail in Sections 1.4.2 and 1.4.4.

1.2.3 Inbound-Outbound traffic analysis

With a *state table* (explained in Section 1.2.2) introduced into the system, all aspects of traffic analysis will have changed with respect to a *stateless* filter. Therefore, there is a significant difference between incoming (*ingoing*) and outgoing (*egress*) traffic. In a *stateless* firewalls, "ingoing" and "egress" are merely topological concepts which require a pair of symmetric rules: to enable a TCP session from inside (the *trust* zone) to outside (the *un-trust* zone), an administrator has to define both an outgoing rule to permit the initiation of the session and an identical mirror rule on the ingoing side to permit the return traffic.

Stateful design, on the other hand, uses session initialization as the basis for defining "ingoing" and "egress." As a result, the firewall analyzes traffic in an asymmetrical manner based on the direction of the first packet (`NEW`). The majority of authorization is provided for the creation of new sessions for *egress* traffic (from a trusted zone toward an untrusted zone.) The *Filtering Policy* (*slow path*) is therefore written mostly to verify the outgoing `NEW` traffic. After a session has been verified and entered into the *state table*, the return

traffic (*ingress* traffic) is automatically categorized as either `ESTABLISHED` or `RELATED`. Therefore, its return traffic is not analyzed against the static *Filtering Policy*, rather the return traffic is routed through the *fast path* via the *state table*.

This method provides **asymmetric rules** and represents one of the primary operational advantages: the administrator only defines the intent (e.g., "allow HTTP from inside to outside") and the firewall dynamically controls the return flow. This greatly simplifies policy and dramatically lowers the attack surface by removing the need for "permissive" static rules to allow the return flow of traffic; which can easily be used by malicious packets to exploit.

In addition to optimizing legitimate traffic, this model is also critical for **management of unknown or anomalous packets**. An `invalid` packet is a packet which appears to have valid headers (for example, an `ACK` or `FIN` flag), yet does not appear to be part of any `NEW` connection being initiated nor does it appear to be part of an existing `ESTABLISHED` session contained in the *state table*. A *stateless* firewall may incorrectly forward an `invalid` packet because it evaluates only the 5-tuple and there exists a generic permissive rule for return traffic. A *stateful* firewall, however, recognizes the contextual conflict (a "lost" `ACK`) and immediately discards the packet, thereby preventing *spoofing* attacks or advanced network scanning.

Finally, the stateful analysis mechanism has to address the issue of ***half-open* connections**. A *half-open* connection arises when a session is terminated abnormally (for example, an endpoint fails without sending `FIN` or `RST` packets) and the terminating endpoint (or the firewall itself) receives no indication of the termination. The allocation for the connection in the *state table* will then remain allocated ("orphaned") and will consume memory resources unnecessarily. To prevent this from occurring, the mechanism for management of inbound/outbound traffic utilizes inactivity timeouts (introduced in Section 1.2.2). If no packet is detected (either inbound or outbound) for a predetermined time (protocol specific), the session is defined as "stale" and the entry in the *state table* is forcibly deleted, thus maintaining the integrity and availability of the *state table*.

1.2.4 Main components

The previously discussed packet flow and operational principles of the stateful firewall (Sections 1.2.1, 1.2.2, 1.2.3) have been implemented through the interactions of the three primary logical architectural elements of the stateful firewall: the **Packet Filter**, the **Connection Tracker** and the **Policy Engine** [14].

The **Packet Filter** is the core element of the firewall and serves as the executive engine of the firewall responsible for making the final decisions regarding the direction of forwarded packets. For stateful firewalls, the Packet Filter has two roles. First, it controls the forwarding of packets corresponding to the *fast path* (i.e., established or related sessions) and acts upon the rapid decision made in the state table during the packet filter's search of the state table. Second, it implements the policy decisions created by the **Policy Engine** for the *slow path* packets (i.e., `NEW` sessions), and executes the final actions taken by the Packet Filter (for example, `accept`, `drop`, `reject`). Thus, it can be considered as the "executive arm" of the entire architecture.

The **Connection Tracker**, or the *State Inspection Engine*, provides the memory

for the firewall and adds context to the firewall. This module is responsible for the complete management of the state table (see Section 1.2.2). One of its key tasks is to capture and inspect every incoming packet, and perform the first search of the state table for each captured packet. Depending on the results of the initial search, the Connection Tracker categorizes the captured packet into one of the four possible states: `NEW`, `ESTABLISHED`, `RELATED`, or `INVALID`. If the captured packet corresponds to an existing session (`established/related`), the Connection Tracker allows the packet to travel via the *fast path*, and updates the session timer. When the captured packet is `NEW`, the Connection Tracker sends the packet to the **Policy Engine** for a deeper inspection. If the packet is `INVALID`, the Connection Tracker discards it. The Connection Tracker is also responsible for creating new entries and deleting expired ones (through either *timeouts* or explicit `FIN/RST` closures).

Lastly, the **Policy Engine** is the decision-making "brain" for the *slow path*. This module is used solely when the **Connection Tracker** forwards a `NEW` packet to the Policy Engine. The only responsibility of the Policy Engine is to enforce the static security policies (i.e., *filtering policy*) configured by the administrator. Therefore, the Policy Engine analyzes the `NEW` packet against the collection of security policies, and makes the determination of whether the beginning of the new session should be allowed (action `ALLOW`) or rejected (actions `DROP` or `REJECT`). Upon completion of the evaluation, the Policy Engine notifies the Connection Tracker of its decision. If the decision is to allow, the Connection Tracker proceeds with the creation of an entry in the state table.

Separating these responsibilities is critical; the **Connection Tracker** manages the state and categorizes packets, the **Policy Engine** enforces rules on `NEW` connections, and the **Packet Filter** physically makes the forward or block. The specific rule structures enforced by the Policy Engine will be examined in detail in Section 1.3.

1.3 Filtering policies and rules

Sections before described the operation design of a **Stateful Firewall**; identified the **Policy Engine** as the logical component to check each connection (via the *Slow Path*) from the start of each new connection (i.e., `NEW` packets) previously defined in Sections before.

This section defines the **Policy Engine**'s main structural components: the **Filtering Policy** (*Filtering Policy*).

The **Filtering Policy** is the formal representation of all static rules defined by an administrator to express the desired level of security for the network. The main role of a *Filtering Policy* in a Stateful Architecture is to determine whether the start of a new session (a.k.a. the `NEW` packet) is allowed or disallowed. A *Filtering Policy* serves as the rulebook for the **Policy Engine** when making the most difficult decisions (i.e., new connections). All other traffic (i.e., `ESTABLISHED/RELATED`) is managed by the **State Table** and thus processed through the *Fast Path*.

In order to be able to completely describe how the **Policy Engine** evaluates the `NEW` packet against the *Filtering Policy*, the next step would be to break down the *Filtering Policy* into its basic elements. Therefore, Section 1.3.1 will thoroughly explain

the syntax of an individual rule in terms of its basic fields (e.g., source, destination, protocol, etc.), followed by an explanation of the logical matching process and resulting actions (Section 1.3.2). Finally, Section 1.3.3 will provide examples of common rules for some important protocols, including TCP and UDP.

1.3.1 Structure of filtering rules

The *Filtering Policy* (Section 1.3's Filtering Policy) consists of a material ordered collection of **filtering rules** (*Filtering Rules*) each representing an atomic command that the *Policy Engine* will use to check whether a `NEW` packet should allow the beginning of a new session.

From a structural perspective, a rule is made up of two basic logical components: a set of **conditions** (*matchers*) and a single **action** (*target*). The action will be performed only when a packet meets all of the conditions stated within a rule.

Fields (*Matchers*) that represent the conditionally based on the packet headers at the L3 and L4 layers are generally what the primary fields are that will be used to define the conditions for a rule. Fields that are included with each implementation may vary; however, the general fields include:

- **Source**: The source IP address or subnet (i.e.: `192.168.1.0/24`) from which the packet originated.
- **Destination**: The IP address or subnet to which the packet is being sent.
- **Protocol**: The transport protocol being used (TCP, UDP, ICMP etc.).
- **Port(s)**: The source and destination ports (port 80 for HTTP) are necessary to identify the specific service.

Rules can also include additional conditional-based fields, including the input/output interface, TCP flags (`SYN`), and in the case of stateful firewalls, the connection state (the connection state is primarily managed by the *Connection Tracker* as described in 1.2.4).

Another key feature that defines how the policy is structured is the order in which the *Policy Engine* evaluates the rules. The rules are not a set, they are a *sequence* (list or chain). When evaluating packets, the *Policy Engine* compares the packet against the rules in a pre-defined sequential order (based upon the rule's priority, often represented by a numerical index). The *Policy Engine* applies a *first-match* principle: once a packet satisfies all of the conditions of a rule, the action associated with that rule is executed (`ACCEPT` or `DROP`), and in most configurations, the processing of the chain for that packet terminates immediately.

The ability to implement sophisticated concepts like **policy inheritance and priority** exists because of the sequential nature of the policy structure. Often, rules are grouped into separate "chains" (*Chains*) (for example, chains that filter `INPUT`, `FORWARD` or `OUTPUT` traffic). A rule in a primary chain can have as its action to "jump" (*Jump*) to a secondary, user created chain. The Packet is then evaluated against the rules of the new chain. If none of the rules in the secondary chain produce a final decision, the packet "returns" to the next rule in the original chain. This *jump* and *return* capability provides a hierarchical

approach to developing complex and modular policies, where one chain inherits from or is subordinate to another.

1.3.2 Packet matching and decision

The logic used to decide what to do with a packet is called **Packet Matching Logic** — a function executed by the *Policy Engine* (Section 1.2.4) to select the appropriate response to a packet.

Packet Matching Logic is a deep packet inspection process that is utilized almost entirely on packets that start a new connection (`NEW`), therefore, all other types of packets will follow the faster path (Section 1.2.3).

The matching process starts when the *Connection Tracker* passes a `NEW` packet to the *Policy Engine*. The *Policy Engine* then evaluates the header fields (specifically, the 5-tuple plus some Layer 2 and/or Layer 3 information including the source interface) of each incoming packet in turn against the *Filtering Policy* in the order specified by the priority order of the rules (Section 1.3.1). The process continues until either of the following occur:

- **First-Match**: The incoming packet matches all the conditions (or matchers) of a particular rule — at that point, the evaluation process ends and the *Policy Engine* will apply the action (target) defined in the matched rule.
- **No-Match**: After evaluating every condition (or matcher) in the policy, there was no single condition that satisfied the requirements — after passing through the entire sequence of rules, the packet reaches the last element in the sequence — the **default action** associated with the firewall is applied (for example, `DROP` in a whitelist configuration).

At the completion of the matching process, the *Policy Engine* can perform one of three primary actions (targets):

- `ACCEPT` (or `ALLOW`): This is the final action — it permits the packet. This instructs the *Packet Filter* (the executive module) to pass the packet to its destination.
- `DROP` (or `DENY`): This is a final action — it silently discards the packet. The firewall deletes the packet from consideration without notifying the sender of the packet deletion. This method of handling packets is generally more secure than others because it does not provide an attacker with evidence of the firewall's presence or configuration during a scan (*security through obscurity*).
- `REJECT`: This is a final action similar to `DROP`; however, the firewall generates an error message to the sender (usually an ICMP *Destination Unreachable – Port Unreachable* message). Although it is more polite and better suited for network troubleshooting, this type of action violates the principle of *security through obscurity* because it notifies an attacker that the host is active and being protected by a firewall.

It is crucial to understand the impact that each action has on both the state of the firewall and future decisions. These decisions, made during the processing of a `NEW` packet,

represent the core of the stateful mechanism. The `DROP` and `REJECT` actions have no stateful implications — they simply end the examination of the current packet with no residual effects on the firewall's memory. On the other hand, the `ACCEPT` action has a significant residual effect — when the *Policy Engine* accepts a `NEW` packet, it does not only permit passage of the packet — it also tells the *Connection Tracker* to add a new record to the state table. That single decision regarding the *slow path* influences all future decisions related to that flow — by adding the state entry to the table, the firewall effectively grants authorization for return traffic (`ESTABLISHED` and `RELATED`) to utilize the *fast path*, thereby bypassing the *Policy Engine* and providing efficiency and asymmetry to the rules as previously discussed (Section 1.2.3).

1.3.3 Practical examples of rules

The conceptual elements of the stateful and decision making models (as described in sections 1.3.1 and 1.3.2) can be implemented as tangible firewall configurations which are dependent upon the transport protocol (Level 4, L4) being filtered. In addition to demonstrating the conceptual model's operational viability, the stateful model's viability in practice is demonstrated through its application to **TCP**, **UDP**, and **ICMP**.

For **TCP**, a connection oriented protocol, the stateful model excels in monitoring the entire session lifetime, where the stateful firewall has the ability to track the entire session lifetime for each user. As discussed in section 1.3.2, a typical *filtering policy* includes rules that do not include all phases of the *handshake*, but instead focus on a permissive rule for the *slowpath* (i.e., `ACCEPT proto=TCP dest-port=80 state=NEW`). If a `SYN` packet (the only packet that matches `state=NEW`) is accepted under this rule, the *Policy Engine* allows the `ACCEPT` action and, as a side effect (section 1.3.2), the *Connection Tracker* inserts a new entry in the *state table*. All subsequent packets (including the `SYN-ACK` response and the final `ACK` of the *handshake*), along with the entire bi-directional dataflow are classified as either `ESTABLISHED` or `RELATED` and therefore processed via the *fast path*.

For **UDP**, a connection-less protocol, the stateful model must simulate the concept of a state since there is no *handshake* to create a state. Therefore, any **UDP** packet that matches an `ACCEPT` rule (i.e., `ALLOW proto=UDP dest-port=53`) is classified as `NEW` and causes an entry to be created in the *state table*. The firewall expects a response packet (classified as `RELATED`) that reverses the 5-tuple (i.e., from a DNS server on port 53). The response packet will be allowed in via the *fast path*. Due to the lack of explicit termination packets, the removal of these state entries is dependent solely on inactivity *timeouts*.

Finally, the treatment of **ICMP** also illustrates the power of `RELATED` tracking. An `ICMP Echo Request` (*ping*) packet that matches an `ACCEPT` rule will cause the creation of a temporary state. The `ICMP Echo Reply` response packet, although technically a new incoming packet, is properly identified by the *Connection Tracker* as `RELATED` to the original *ping* and allowed via the *fast path*. Another example is an `ICMP Destination Unreachable (Port Unreachable)` packet that is transmitted by a router in response to a failed **UDP** attempt; a *stateless* filter would drop it, while a stateful firewall classifies it as `RELATED` to the **UDP** session (that is still present in the *state table*) and transmits it to the sender.

To demonstrate how these concepts are converted into actual configurations, Listing 1.1

presents an example of a ruleset that uses a common syntax (similar to `iptables`) and implements a *whitelisting* policy (Default `DROP`):

Listing 1.1: Example of stateful ruleset (iptables syntax)

```
# 1. Enable "Fast Path" for all already known session traffic.
# This is the most important rule for performance.
-A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT

# 2. Enable "Slow Path" (NEW) for specific TCP sessions (e.g., Web)
-A FORWARD -p tcp --dport 80 -m conntrack --ctstate NEW -j ACCEPT
-A FORWARD -p tcp --dport 443 -m conntrack --ctstate NEW -j ACCEPT

# 3. Enable "Slow Path" (NEW) for specific UDP sessions (e.g., DNS)
-A FORWARD -p udp --dport 53 -m conntrack --ctstate NEW -j ACCEPT

# 4. Enable "Slow Path" (NEW) for ICMP (e.g., Ping)
-A FORWARD -p icmp --icmp-type echo-request -m conntrack --ctstate NEW -j ACCEPT

# 5. Manage anomalous packets (discarded for security)
-A FORWARD -m conntrack --ctstate INVALID -j DROP

# 6. Default Action (Whitelisting): everything that is not
# ESTABLISHED, RELATED or NEW (permitted) is discarded.
-P FORWARD DROP
```

As highlighted in Listing 1.1, the *Filtering Policy* focuses almost exclusively on defining which `NEW` packets are authorized to start a session. The first rule (`ESTABLISHED,RELATED`) then delegates the management of almost all traffic (the responses and data flows) to the state table, implementing the efficiency of the *fast path*.

1.4 State management and connection tracking

This section is an analysis of the mechanisms used by firewalls to track and maintain sessions (Section 1.2.4), and therefore how it manages its "memory" of sessions, which will allow us to assess whether this mechanism works well or not.

In particular, we will investigate how the *Connection Tracker* maintains the *state table*, but not only how new records are created (Section 1.4.1); we will focus on the lifecycle of the sessions and the role of the *timeout* (inactivity timer), that can be different depending on the protocols used (TCP/UDP), so that the orphaned connections can be automatically deleted (Section 1.4.2);

We will finally highlight the differences in operation between these firewalls and those using a *stateless filter* (Section 1.4.3), and the performance issues they entail; finally, we will also highlight the internal problems of this mechanism, the risk of saturating the state table (it represents a vector of denial-of-service attacks) and some possible optimizations and techniques of *garbage collection* (Section 1.4.4).

1.4.1 Connection tracking and state tables

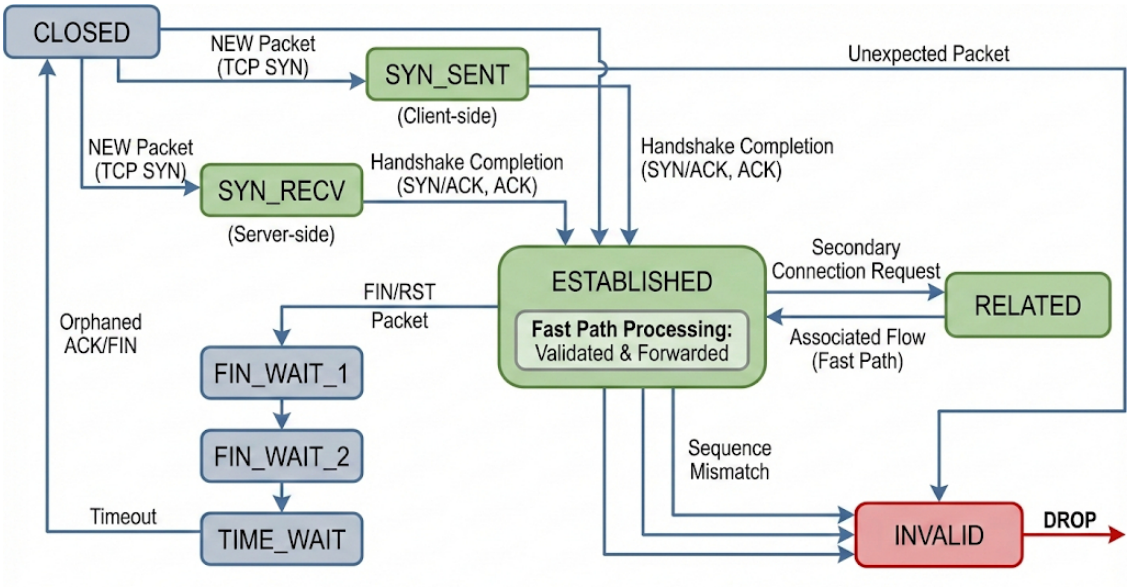


Figure 1.2: TCP Transition State Diagram

Connection tracking is the core process that enables a stateful firewall to manage its state (discussed in Section 1.4), and is much more than simply entering an entry in the state table (covered in Section 1.2.2). It is essentially the semantic engine behind the scene that tracks the life cycle of sessions (traffic flows) as a real finite state machine for each one [14], and that allows the firewall to distinguish between the various states of a connection, and categorize every packet into specific operational categories.

The ultimate objective of connection tracking is to determine the **relationship between packets and connections**. A packet is more than just a 5-tuple datagram; it is an event occurring in a logical sequence. The *Connection Tracker* classifies packets into four fundamental states:

- **`NEW`:** Identifies packets trying to create a new connection. There is no related entry in the *state table* for `NEW` packets. `NEW` packets are generally the case of TCP `SYN` packets, or the first packet in a UDP flow. All `NEW` packets are processed by the *slow path* and are evaluated by the *Policy Engine* (seen in 1.2.1 and 1.3.2). If the Policy permits the packet, the *Connection Tracker* creates and manages the state entry, setting the first state.

- **`ESTABLISHED`:** Identifies packets that are part of an existing and being tracked connection in the *state table*, in either direction. Once the connection is confirmed (for example, once the TCP *three-way handshake* is complete), packets enter the `ESTABLISHED` state. `ESTABLISHED` packets are processed entirely on the *fast path*: they undergo validation (for example, checks of consistency in TCP sequence numbers) and are rapidly forwarded without evaluation by the *Filtering Policy*.

- **`RELATED`:** Identifies packets that do not belong to the original connection (the 5-tuple of the packets differs), but are logically and algorithmically linked to it. This is an important state for complex protocols. Classic cases of `RELATED` packets are an ICMP *Destination Unreachable* packet responding to a blocked UDP packet, or the secondary data connection of a protocol like FTP, which is "related" to the primary control connection. Like `ESTABLISHED` packets, `RELATED` packets are also processed on the *fast path*.

- **`INVALID`:** Identifies packets that belong to none of the above states. Generally speaking, these are packets that could never begin a new session (for example, orphaned `ACK` or `FIN` packets with no existing connection) or those whose consistency checks have failed (for example, obviously incorrect TCP sequence numbers). These packets are usually dropped (`DROP`) for security reasons, because they can indicate scans or attacks.

1.4.2 Timeout and inactive session management

The overall design element that will ensure the stability and resilience of the state table (which was introduced in Section 1.2.2) is the implementation of **inactivity timers** (also referred to as *timeouts*). Termination of sessions through explicit methods (such as TCP `FIN`/`RST`) is used to end sessions properly, while *timeouts* are used as a *garbage collector* method to remove inactive (and often called "orphaned") sessions automatically; thus helping prevent state table overflow, either from abnormal interruption of connections (i.e., *half-open*) or through connectionless protocols which do not support a defined close message.

There should be no reliance upon a single timeout value when developing a timeout management plan, as it is better to use **protocol-defined timeout parameters**, along with a separate timeout for each connection state for **TCP** (identified using the *Connection Tracker* in Section 1.4.1). Due to the complexity of the state machine maintained by the *Connection Tracker*, the timeouts can be very granular:

- A `SYN_SENT` packet has a very short timer (for example, 30–60 seconds) because this is a key defense mechanism: if the *handshake* is not completed (because of a *SYN flood* attack or a host that is unreachable) the "tentative" state entry will expire quickly so as to prevent the table from becoming saturated by malicious *half-open* connections.

- An `ESTABLISHED` session has a very long timer (for example, many hours or days) since legitimate connections (such as SSH sessions or remote terminals) could continue to exist without activity from the user for long periods of time.

- Closing states (`FIN_WAIT`, `TIME_WAIT`) have unique and short timers as well, so as to allow a session to be closed properly after it is terminated, yet not before any delayed packets have been transmitted.

UDP, however, behaves differently. Since UDP is *connectionless*, the firewall simulates a state for UDP. When a UDP `NEW` packet creates a new state entry, the firewall waits for

a response. As a result, the initial timer for a UDP `NEW` session is usually very aggressive (typically 30 seconds or less). If a related packet (RESPONSE) is received, the UDP state entry is confirmed (some times considered as `ESTABLISHED`) and the timer is extended (for example, 120–180 seconds) and reset with every subsequent packet received. If there is no response, the entry is timed-out rapidly [14].

Finally, **ICMP** (discussed in Section 1.3.3) has its own timers that are relatively short to identify relationships between requests and responses (for example, `Echo Request`/`Echo Reply`).

The **performance and security implications** of this mechanism represent a critical *trade-off*. On the one hand (security), aggressive timers (particularly for `SYN_SENT` and `UDP_NEW`) are a fundamental defensive mechanism against *resource exhaustion* attacks which are designed to fill the state table (the subject of Section 1.4.4). On the other hand (performance and reliability), evaluating millions of active timers repeatedly represents a significant CPU *overhead* (non-negligible) and may disrupt legitimate but temporarily inactive connections. Therefore, optimal configuration of *timeout* parameters are necessary to achieve an acceptable level of both security and usability for a service.

Typical timeout values for some of the most common states and protocols are summarized in Table 1.2.

Protocol State	Protocol	Timeout (s)	Motivation
`TCP_SYN_SENT`	TCP	60	SYN Flood mitigation, rapid closure of failed attempts
`TCP_ESTABLISHED`	TCP	432 000 (5 days)	Support for long-lived sessions (e.g., SSH)
`TCP_FIN_WAIT`	TCP	120	Waiting for orderly closure (final `ACK`)
`UDP_NEW`	UDP	30	Rapid removal of scans or one-shot UDP traffic
`UDP_ESTABLISHED`	UDP	180	State maintenance for active UDP flows (e.g., streaming)
`ICMP_REQUEST`	ICMP	30	Short wait for a response (e.g., *ping*)

Table 1.2: Timeout connection examples for TCP

1.4.3 Differences from stateless firewalls

While prior sections covered the conceptual differentiation between stateless and stateful filtering (Section 1.1.2), this is where those distinctions become evident after the discussion of state table management (Section 1.4.1) and timeouts (Section 1.4.2) is completed. The differences between the two types of firewalls go beyond functional differences; they are also in how decisions are made about what to do with each packet and the associated **performance** and resource consumption.

Stateless firewalls operate in a totally "atomic" fashion. Each packet is treated independently and is compared to the complete and static *Filtering Policy*. Whether a packet is `ACCEPT`ed or `DROP`ped is based upon the content of the packet itself. **Stateful firewalls**, however, make decisions based upon **context** and **history**. They utilize a two-speed logic (as detailed in Section 1.2.1) to evaluate packets: the first packet in a given session (`NEW`) is analyzed in detail (*slow path*) and all other packets in the same session (`ESTABLISHED`) are quickly examined (high speed) via a state table look up ($O(1)$).

In addition to the decision making process, this architecture allows firewalls to keep track of packets over time. Stateless filters have no "memory." Therefore, when an incoming `ACK` packet arrives, there is no way to know if the `ACK` packet is actually the reply to a `SYN` packet sent several minutes ago or an invalid packet as part of some type of malicious scan. Stateful firewalls, however, can correlate packets semantically. They identify response traffic (`ESTABLISHED`) and secondary flows (`RELATED`, i.e., ICMP or FTP data) since they have previously identified the `NEW` session from which those flows originated.

These differences in operation have a direct relationship to *throughput* and resource utilization. Ironically, even though a stateful firewall performs a more complicated evaluation, it will typically have higher *throughput* than a stateless filter. Stateless filters reevaluate the entire *ruleset* (which may consist of thousands of rules) for every single packet in a session. Stateful firewalls pay this computational (CPU) price only one time per session (*slow path*) and the remaining 99% of the packets in a session are processed very rapidly (*fast path*).

However, there is a resource price for this increased throughput: the **memory** needed to maintain the *state table*. The state table is a significant bottleneck since it needs to be large enough to store all current sessions and fast enough so as not to diminish the performance of the system. However, the state table itself is vulnerable to *resource exhaustion* attacks, as explained in the following section (Section 1.4.4).

1.4.4 Problems and optimizations in state management

Although the management of the state table provides the functionality which makes stateful firewalls so superior (as discussed in Section 1.4.3), it also presents an inherent weakness: **table saturation**. Table saturation is the major performance limiting factor and a key target for attackers of stateful devices. Although tables have limited memory (and therefore represent finite-sized data structures), they are vulnerable to *Denial of Service* (DoS) attacks intended to exhaust their capacity (i.e., *resource exhaustion*). By sending a large number of `NEW` packets with spoofed source IP addresses, an attacker can mount a DoS attack (e.g., *SYN flood*). Because the firewall must create a table entry for each of the `NEW` packets it receives (in accordance with Section 1.4.1), the number of entries in the table increases rapidly until it is full. At that point, the firewall cannot add new entries to the table and thus cannot allow any legitimate new connections to be initiated.

In order to reduce this basic threat, there are various mechanisms that have been designed to optimize and provide *garbage collection* for the state table. The first line of defense for the garbage collection (cleanup) processes were explicitly closed connections

(`FIN`/`RST`) and inactivity timeouts (Section 1.4.2). The use of short (aggressive) timers, particularly for the `SYN_SENT` and `UDP_NEW` states, is a direct countermeasures to prevent saturation attacks; thereby ensuring that "tentative" or *half-open* entries are deleted quickly when the connection has completed. While garbage collection is effective, efficient management of millions of connections requires that additional optimizations be applied to the data structure.

The primary optimization technique is the use of **hashing**. In addition to being mentioned in Section 1.2.2, the *state table* is implemented as a *hash table* to ensure that lookups of packets in the *fast path* (which are the majority of operations) occur in constant time, $O(1)$, and do not become a bottleneck for table access [14].

Aging is the process used to implement timeouts. The system must either periodically scan the table to find and delete timed-out entries, or utilize other data structures (such as *timer wheels*) to perform this function. However, there is a *trade-off* between the frequency of scanning, and the impact of the CPU, and between the frequency of scanning, and the latency of reclaiming memory.

Finally, in *High Availability* (HA) and distributed architectures, the necessity for **clustering** arises. To ensure redundancy, the state of a connection managed by a "primary" firewall must be duplicated to a "secondary" (or *stand-by*) firewall. This creates a problem of synchronizing state between multiple nodes, which will be studied in detail in Section 1.6.2.

1.5 Common implementations and technologies

In the preceding sections (Sections 1.1–1.4) the theoretical framework for *Stateful Firewalls* has been established; specifically, the conceptual model, logical design and the operational fundamentals that define how they function have been described. These descriptions were based upon theoretical models of mechanisms commonly used in all *Stateful Firewall* implementations including *connection tracking* and the management of the *state table*.

The next portion of this report focuses upon the practical application of the previously stated abstract concepts. Specifically, it addresses how each of the theoretical constructs and principles are implemented within specific state-of-the-art products and technologies. An understanding of various product implementation approaches can assist in identifying the inherent *trade-offs* in terms of *performance*, cost, flexibility and integration.

An overview of representative products of *stateful firewalls* (Section 1.5.1) will be provided first. The overview will examine platforms that have set both industrial and open source standards; for example, `iptables`/`nftables` (linux), `pf` (openbsd) and commercial appliance vendors, i.e., Cisco [15]. Following the overview of well known implementations, the focus will shift to the significant architectural differences between the two categories of stateful firewall implementations (hardware) versus those that are exclusively based upon software (Section 1.5.2). The *performance* benefits and disadvantages of *general purpose* architectures versus those that are specialized will be addressed. In conclusion, the final segment of this section will address the integration of these technologies in virtualized and containerized environments (Section 1.5.3), a deployment environment in which, as identified in Section 1.1.1, both *performance* and isolation characteristics present unique

challenges in *software defined* infrastructure.

1.5.1 Examples of well-known stateful firewalls

The methods used for implementing connection tracking (Section 1.4.1) and time-outs (Section 1.4.2) of firewall products exist in a broad variety of forms within both commercial and open-source firewalls. Each of these implementations have similar goals (keeping track of state), but each has very different design philosophies and architectures.

The majority of open-source implementations for maintaining a state-based firewall are based upon the historical packet filtering mechanism of the Linux kernel called `iptables`. However, unlike the packet filtering mechanism of `iptables` that maintains the state of packets it is filtering, the ability of `iptables` to maintain state is a separate module known as `conntrack` (connection tracker). `conntrack` is also a kernel-based subsystem that intercepts packets before they reach `iptables` to look up the corresponding entry in the state table, and marks the packets with one of five possible states (`NEW`, `ESTABLISHED`, `RELATED`, `INVALID`). Packet filtering rules may then utilize these states to enable the *fast-path* of traffic through the firewall. The modern replacement for `iptables`, `nftables`, provides the same functionality of performing state-table lookups and marking packets with state information; however, does so in a more cohesive manner and at greater speeds than its predecessor [14].

As an alternative to the development of a separate module for maintaining state in `iptables`, the developers of `pf` (packet filter) chose to integrate the ability to create state into the native rule syntax of their firewall product. This approach is often referred to as *stateful inspection*, and `pf` was developed as a replacement for the previous firewall product for OpenBSD known as `IPFilter`. `pf` has received recognition for its clean, readable, and powerful rule syntax (*rule set*), as well as its reliability. Therefore, rather than relying on a separate module to create and manage the state associated with packets processed through a firewall, in `pf` the process of creating the state of packets processed through the firewall is an integral and inherent part of the native filtering rule syntax [14].

As opposed to the open-source firewall community, the commercial firewall community is dominated by dedicated hardware appliance platforms. Firewalls such as **Cisco ASA** (Adaptive Security Appliance) and **Juniper SRX Series** are examples of this type of platform. Although the concepts of how these platforms operate are virtually identical to those described above (i.e., the operation of processing the 5-tuple of packets, identifying whether packets being processed are new, and providing a fast path to packets that have been previously processed), the platforms' architectures differ significantly from purely software-based systems. Rather than utilizing general purpose CPU's to perform state table lookups, these platforms provide special purpose hardware (ASIC's, NPUs, etc.) to enhance the speed and capacity of the state table lookups. As a result of the existence of this hardware, the platforms are able to support significantly higher numbers of sessions (connections per second) and state tables (millions of entries).

Therefore, the differing design philosophies of the above-mentioned platforms produce a number of different trade-off's that are summarized below in Table 1.3. A primary difference between the two types of platforms (i.e., the platforms operating on generic hardware vs. the platforms operating on specialized hardware) produces a primary

Implementation	Platform	State Mechanism	Approach
`iptables/ nftables`	Linux	Kernel `conntrack` module	SW on *general-purpose* OS
`pf`	OpenBSD, FreeBSD	Integrated in the filter	SW on *general-purpose* OS
Cisco ASA / Juniper SRX	Cisco IOS/ASA, Junos OS	Integrated and accelerated	*HW Appliance* (*purpose-built*)

Table 1.3: Comparison of most widely used stateful firewalls implementation

distinction between software and hardware architectures of firewalls. These distinctions will be examined further in the next section.

1.5.2 Hardware vs. Software Architectures

A *general-purpose* software solution, based on the architecture described in Section 1.5.1, involves using dedicated hardware-based firewalls (specialized *appliances*) versus completely software-based (*general-purpose*) solutions, which will determine the fundamental *trade-offs* in terms of performance, flexibility and cost.

Software architectures, whether they are implementing the *stateful* firewall logic (for example, `iptables/conntrack` or `pf`) as software processes on commodity hardware (COTS), essentially standard *servers* (x86 architecture) provide great flexibility. Software functions can be easily modified, updated and deployed just like any other piece of software. Additionally, initial costs associated with *general-purpose* software architectures are typically lower because they do not require specialized hardware and therefore are less dependent upon the proprietary hardware of a particular manufacturer.

However, *general-purpose* software architectures have significant performance limitations. In addition to the fact that the entire state management process, including the *lookup* in the *state table* for the *fast path*, compete for resources (for example, CPU cycles, memory access, I/O *interrupts*) with the *general-purpose* operating system and all other processes, even though the *fast path* does not pass through the *Policy Engine*, the packet is still required to be processed by the *kernel*'s network *stack*, which imposes an *overhead* that will limit the maximum *throughput* and the number of connections per second that can be managed by the *general-purpose* software architecture.

On the other hand, firewalls that are built into dedicated *hardware appliances* (for example, Cisco ASA, Juniper SRX) are specifically designed for packet processing. Therefore, their internal architecture will differ dramatically from that of a *general-purpose* server. Like the *general-purpose* software architecture, the *control plane* (management, policy, `NEW` session handling of the *slow path*) of a dedicated *hardware appliance* will operate on a conventional CPU; however, the *data plane* (forwarding of `ESTABLISHED` packets on the *fast path*) will be implemented directly in silicon. To enable the *data plane* to operate at *wire speed*, these appliances utilize ASICs (*Application-Specific Integrated Circuits*) and NPUs (*Network Processing Units*) to perform the most time-critical operations (for example, state table *lookup* and packet forwarding).

The benefit of using a dedicated *hardware appliance* is that it provides a level of performance (measured in Gbps of *throughput* and millions of concurrent connections)

that is many orders of magnitude greater than what can be achieved by a *general-purpose* software solution on commodity hardware. There is, however, a cost associated with using dedicated *hardware appliances*. Lower flexibility (the features of the ASIC cannot be changed) and higher purchase prices are two of the most obvious disadvantages. In addition, users of dedicated *hardware appliances* will find themselves to be highly dependent upon the vendor who manufactured the product (a condition known as *vendor lock-in*) [14].

Criterion	**Software Architecture** (*General-Purpose*)	**Hardware Architecture** (*Purpose-Built*)
Platform	COTS Servers (e.g., x86)	Specialized Hardware (ASIC/NPU)
Examples	`iptables` (Linux), `pf` (OpenBSD)	Cisco ASA, Juniper SRX
Performance	Limited by *general-purpose* CPU and *kernel* network stack	*Wire speed fast path* (ASIC accelerated)
Flexibility	High (easy software updates)	Low (functionality tied to hardware)
Cost	Low (non-proprietary hardware)	High (proprietary hardware)
Use Case	SMEs, Virtualized Environments, Flexibility	*Data Centers*, *Service Providers*, High Performance

Table 1.4: Comparison between Hardware and Software Architectures

1.5.3 Integration with virtualized and containerized environments

The Software-Defined Architecture (SDA) paradigm introduced in Section 1.1.1 — by way of both **Network Functions Virtualization** (NFV) and **containers** — provides the main application domain for *software-stateful* firewalls (as discussed in Section 1.5.2). With the SDA paradigm, traditional *purpose-built hardware appliances* become less effective; as the SDA paradigm emphasizes agility, elastic scalability, and rapid provisioning of resources — all of which can only be achieved with *software* solutions.

In the **NFV** paradigm, a *stateful firewall* (for example, an instance of `iptables` or `pf`) is "packaged" and provisioned as a **Virtual Network Function** (VNF). As such, the VNF will run as a VM on top of a *hypervisor* (for example, KVM, VMware), or as a direct guest operating system on a COTS server. A major challenge when using the NFV paradigm is adapting to *software defined* infrastructure paradigms: the traffic must be properly "steered" (typically via *Service Function Chaining*) so that it passes through the firewall VNF prior to being reinjected back into the virtual network [7] where *connection tracking* logic is applied.

As opposed to the NFV paradigm, integrating a stateful firewall into **containerized environments** (for example, Docker, Kubernetes) present different challenges. Containers do not have their own OS kernel, thus, *stateful* filtering within a container typically depends upon the `conntrack` kernel module running on the *host*. Policies (for example, Kubernetes *Network Policies*) are then interpreted by the host as `iptables` or `nftables`

rules, and filter traffic passing through either virtual *bridges* (for example, `docker0`), or `veth` interfaces. While providing *stateful* filtering at the level of individual containers, the integration into containerized environments presents two additional challenges:

1. **Performance:** When using a centralized `conntrack` module in the host kernel to track the state of each of thousands of *containers* (where each container may contain thousands of session states), it becomes a performance *bottleneck*. Additionally, because the single host maintains a single state table, the overhead of managing that state table (as discussed in Section 1.4.4) is amplified, since there is a need to manage a separate state table per *tenant* (container).

2. **Isolation:** When multiple containers share the same host, the shared state table becomes a source of potential isolation issues. Although Linux provides *network namespace* virtualization of the network stack (including `iptables` tables), the `conntrack` module has traditionally been a global resource, allowing for potential interference or *side-channel* leaks between *containers*. Recent improvements to the `conntrack` module (for example, *conntrack zones*) are attempting to address this challenge, however, the overall architecture of maintaining state in a high-density *container* environment continues to be one of the most significant architectural challenges to date.

1.6 Limitations and challenges

Sections 1.1–1.5 of this Chapter have detailed the Conceptual Model, Operational Architecture, Filtering (Policy) Principles and Major Technological Implementations of *Stateful Firewalls*. Thus it can be seen that the ability of Stateful Systems to keep track of Sessions and thus provide Granular and Secure Traffic Control (e.g., Management of *Fast Path* for `ESTABLISHED` sessions) is based on their fundamental capability—stateful session tracking (Section 1.4.1). However, this capability also creates significant computational and memory *overhead*, which represents the **major limitation of the Stateful Paradigm**. Managing thousands or millions of concurrent sessions creates significant Resource Challenges.

In this final section of the Chapter, the two main **Architectural Challenges** introduced by the Management of Session State will be analyzed:

- **Scalability and Performance** (Section 1.6.1), where the Intrinsic Scalability Limits will be discussed, examining again the Concept of the *State Table* as a *Bottleneck* (as described in Section 1.4.4), as well as the Overhead of Connection Tracking on Overall System Performance;

- **State Synchronization** (Section 1.6.2); one of the most Complex Issues in Modern Distributed Architectures (like NFV), specifically the Replication and Synchronization of the State of Connections between Multiple Firewall Instances to achieve *High Availability* and Policy Consistency.

1.6.1 Scalability and performance

Although the stateful architecture offers fine-grained control over network traffic flow, the inherent constraints of the stateful design lead to limitations in terms of performance and scalability due to "inherent" bottlenecks.

The first of these bottlenecks is the **state table** itself, which has the disadvantage of being an array-based data structure of limited size (dependent upon the amount of memory or RAM installed in the device), thereby providing a significant point of attack for *Denial of Service* (DoS) based on resource starvation (e.g., `SYN flood`).

The second bottleneck is the **computational overhead (CPU)** which affects two different metrics:

- **Connection Rate (CPS):** the number of *new* connections/second; CPS is limited by the *slow path*, which is CPU intensive. Each `NEW` packet processed via the *slow path* incurs the cost of evaluating the complete Filtering Policy, plus creating a new entry in the state table.
- **Throughput (Gbps):** the volume of data which can be processed through the system; while the *fast path* for `ESTABLISHED` sessions has been optimized, processing each `ESTABLISHED` session still requires a *lookup* operation and a write to update session timers, both of which consume CPU cycles.

In addition to the above mentioned factors, the problem is further complicated when operating in virtualized or containerized environments as described in Section 1.5.3. In such cases, a centralized `conntrack` module within the host kernel may act as a shared bottleneck limiting throughput for all *containers* running on the same host.

As such, the only way to overcome the maximum **vertical scalability** limit of a single firewall instance (i.e., adding more CPUs and RAM to a single machine) is to implement **horizontal scalability** by dividing the workload among multiple parallel firewall instances, which are typically load balanced together. However, this approach raises the complexity of maintaining symmetric routing and state consistency between the various nodes involved, which will be addressed in Section 1.6.2 [7].

1.6.2 State synchronization issues between multiple nodes

If the scalability challenges described in Section 1.6.1 are solved with **horizontal scaling** (i.e., by creating many parallel firewalls that are each managed by a load balancer), then you will immediately run into the most difficult architectural problems of *stateful* systems: **State Synchronization**.

State Synchronization arises because of *Asymmetric Forwarding*. A `NEW` packet (a packet that goes down the *slow path*) could be routed to Firewall-A (by the load balancer) that Firewall-A uses to create a state entry in its own table. But when the response packet arrives, that packet belongs to `ESTABLISHED` state and gets routed to Firewall-B. Because Firewall-B doesn't know anything about the state entry (the state entry only exists on Firewall-A), Firewall-B will incorrectly classify the packet as `INVALID` (because it does not match the definition of Section 1.4.1) and drop it. Dropping the packet will terminate

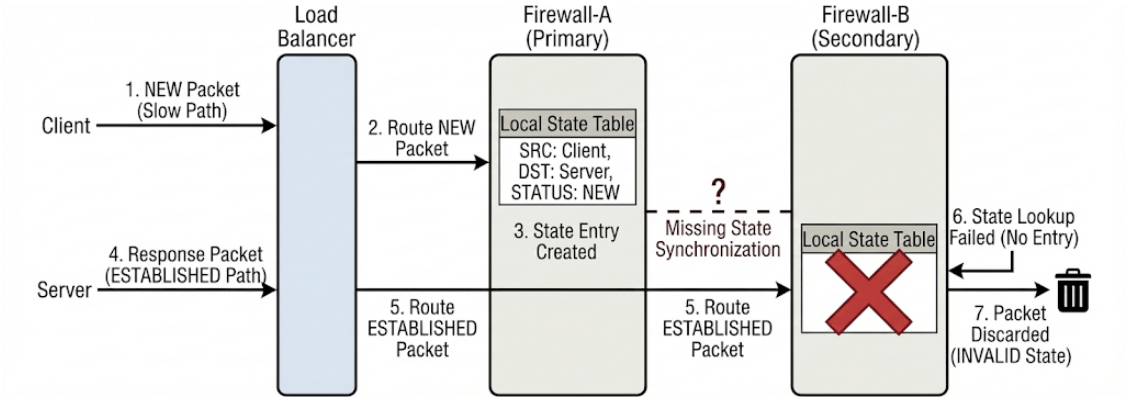


Figure 1.3: State failure diagram due to asymmetric forwarding in a firewall cluster. *Asymmetric forwarding leads to Firewall-B receiving a response packet for a connection tracked only on firewall-A, causing legitimate traffic drop and connection failure.*

a valid connection. This is especially serious in *High Availability* (HA) situations. If Firewall-A fails (a *failover* occurs), all million+ connections that it has active at the time will be terminated because the backup Firewall-B will have no record of those states.

As a result, there needs to be an **Active Synchronization mechanism** implemented between all the nodes in the cluster so that whenever Firewall-A adds a new state entry or updates an existing one, Firewall-A must notify all of the peer nodes (B, C, etc.) of this addition/update. Normally, the notifications will happen over a high speed dedicated *backplane* network. While this provides for a solution to the problem, it also generates a lot of CPU usage and bandwidth consumption that can easily become resource intensive and negate any performance advantages gained by horizontally scaling the firewall(s).

Additionally, the distributed context of a cluster presents additional complexity (consistency) issues. Any synchronization must be done with less delay than the round trip time of the network traffic; otherwise, a response packet (`ESTABLISHED`) could reach Firewall-B before Firewall-A sends the synchronization message to Firewall-B and both could end up terminating the session. Maintaining consistency in a distributed system is known to be a very difficult task, maintaining the consistency of the timing of events in a distributed system (*race conditions*) and managing timeouts are just two of the many tasks [7].

These practical constraints are made even worse in distributed environments like **NFV** (explained in Sections 1.1.1 and 1.5.3). The elasticity of the environment of NFV, as described in sections 1.1.1 and 1.5.3, enables the creation and termination of VNFs (firewall instances) to *scale out*.

1.7 Conclusions

This Chapter provides a technical and conceptual examination of **stateful firewalls**, providing a basis for the thesis. From the current state of modern network architectures (Section 1.1), it is demonstrated how manual configuration and the need for formal

automation is significant. In particular, the difference between the operation of *stateless* versus *stateful* filters (Section 1.1.2) is identified; the **state table** is the primary element that enables a firewall to inspect traffic within a context.

From there, the architecture (Section 1.2) is broken down into its logical components, including the **Connection Tracker** and **Policy Engine** and the concept of a two-speed operational model (*slow path* and *fast path*). The Analysis continues with the examination of filtering Policies (Section 1.3) and connection tracking methods (Section 1.4) focusing on the differences in protocol handling (TCP, UDP, ICMP) and the significance of inactivity timeouts. The typical technological implementation (Section 1.5) and, more importantly, the fundamental limitations of the stateful paradigm (Section 1.6) have also been evaluated.

In particular, the limitations — especially performance overhead and scalability (Section 1.6.1) — that exist with respect to modern network architectures are exacerbated. Today's networks, which feature both virtualization (NFV) and containers, and service graphs that are increasingly complex are highly dynamic: topologies change rapidly (*scale-out*); redundant architectures (HA) are used and managing the complexities of distributed flow configurations, many of which are asymmetrical, are difficult to manage manually.

Therefore, the traditional form of security management (introduced in Section 1.1.1) is unworkable. Maintaining synchronization between multiple distributed topologies, which are constantly changing and are subject to workarounds specific to their topology, is very difficult and expensive. Managing the distribution of policies is error prone and is typically managed with a "it probably works" configuration without formal guarantees.

It is therefore necessary to transition from a manual management process — i.e., the fallible and reactive "probably correct" configuration — to an automated process — i.e., an automated "provably correct" configuration. An appropriate solution is required to shift the complexity from fallible and reactive manual management to a process that is formally correct — i.e., *correctness-by-construction*.

Chapter 2 will present the **VEREFOO** framework, the formal methodology at the core of this thesis, developed using MaxSMT formulations to meet the challenges described here.

Chapter 2

The VEREFOO Approach

As described above, Chapter 1 was a comprehensive analysis of *stateful firewalls* (both technically and conceptually) which provides the background for this thesis work, and identifies the intrinsic limitations of stateful firewalls (Section 1.6). Of particular interest were two architectural limitations identified in Chapter 1: the management-based performance *overhead*, and the potential state table saturation (Sections 1.4.4 and 1.6.1) and the substantial complexity of state synchronization in real-time (Section 1.6.2). As noted in Section 1.6.2, particularly in distributed and dynamic architectures such as those found in NFV, the state synchronization challenge makes both manual configuration and traditional management paradigms based on *runtime* state less than ideal, complex, and error-prone.

Therefore, it follows that the challenges discussed above necessitate an abstraction level of thinking to shift the complexities associated with *runtime* management to a formal design and orchestration phase. Chapter 2 presents the **VEREFOO** (*Verified Refinement and Optimized Orchestration*) *framework* [6, 7] - the formal methodology underpinning this thesis and developed to automatically allocate and configure **Network Security Functions (NSFs)** within virtualized networks; i.e., to create globally correct and optimal security configurations (*correctness-by-construction*) that eliminate the need for manual management. This chapter further describes the VEREFOO *framework* as follows:

Section 2.1 will examine the broader context of SDN/NFV architectures, with emphasis placed on the motivation behind the use of formal automation for the management of security configuration.

Section 2.2 will formally describe the models upon which VEREFOO relies for its reasoning processes: how the *Service Graph* and the *Allocation Graph* represent the network topology, how traffic flows can be represented (**Maximal Flows** [5]), and how **Network Security Requirements (NSRs)** can be formally expressed.

Section 2.3 will describe the specific interface through which users interact with these formal models and how user specifications are translated to formal models using XML *input* schemas and the *parsing* process employed by VEREFOO.

Section 2.4 will explain the core resolution mechanism of VEREFOO: the formulation of the *refinement* problem as a MaxSMT instance, and define the structure of the constraints (*Hard* and *Soft*).

Finally, Section 2.5 is the conceptual core of this thesis and will describe the extension of VEREFOO to provide the semantics of *stateful firewalls*: including the `ALLOW_COND` action and the logic of conditional permissions.

Section 2.6 concludes the description of VEREFOO, discussing the implications and theoretical advantages of the VEREFOO *framework* (distributed security management and scalability), and sets the stage for the experimental evaluation and verification presented in subsequent chapters.

2.1 Context and Motivations for Automation in NFV

The architectural limitations of traditional stateful firewalls were addressed in Chapter 1, as they have been identified to be particularly limited with regards to scalability and state synchronization when distributed (Section 1.6). These limitations are a result of the shift from static, hardware based network configurations to virtualized infrastructure models [7] which provide the base for this dissertation work.

Before discussing the formal methodology of VEREFOO (introduced in the preamble to this chapter), it is important to define the operational environment and motivation behind why automation is a requirement in these types of environments. Although SDN and NFV are beneficial to the operation of the network in terms of agility and flexibility, they create significant security management problems, thus making manual approaches both obsolete and inherently insecure [6].

Therefore, this section will discuss the context and rationale for adopting a formal and automated approach. This will include a description of the fundamental changes from traditional networks to virtualized architectures (Section 2.1.1) followed by a discussion of the specific security configuration challenges created by these changes (Section 2.1.2).

2.1.1 From Traditional Networks to Virtualized Architectures (SDN/ NFV)

Traditionally, networks were developed around rigidly coupled network components and proprietary hardware (which is often referred to as *middleboxes*), such as routers, load balancers and firewalls. Although robust and reliable, the hardware-based approach was limited in terms of scalability. Any changes to policies for traffic flow, etc. required significant, time consuming and complicated work, and did not lend themselves to the rapid and dynamic service demands of today's environments.

Two new approaches have now arisen which provide the necessary infrastructure for today's rapidly changing environment; **Software Defined Networks** (SDNs) and **Network Function Virtualization** (NFVs).

Software Defined Networking (SDNs) introduce a formal separation between the *control plane* (where all the "intelligence" exists, i.e. the "brain") and the *data plane* (where the actual network processing takes place, i.e. switches, routers, etc.). The result is that the SDN *control plane* acts as a central software *controller*, whereas the network devices act as programmable forwarding executors. Therefore, it is possible to flexibly manage the network through a software-defined method, whereby routing and filtering decisions can be made programmatically by the *controller*.

Network Function Virtualization (NFV), however, addresses the issue of hardware dependency. NFV separates network functions (NF), such as firewalls or NATs, from the specific hardware that they run on [3,7]. Instead of running on dedicated hardware, network functions are implemented as software programs, called **Virtual Network Functions** (VNF), which can run on COTS (*Commercial Off-The-Shelf*) hardware platforms, such as x86 servers [3].

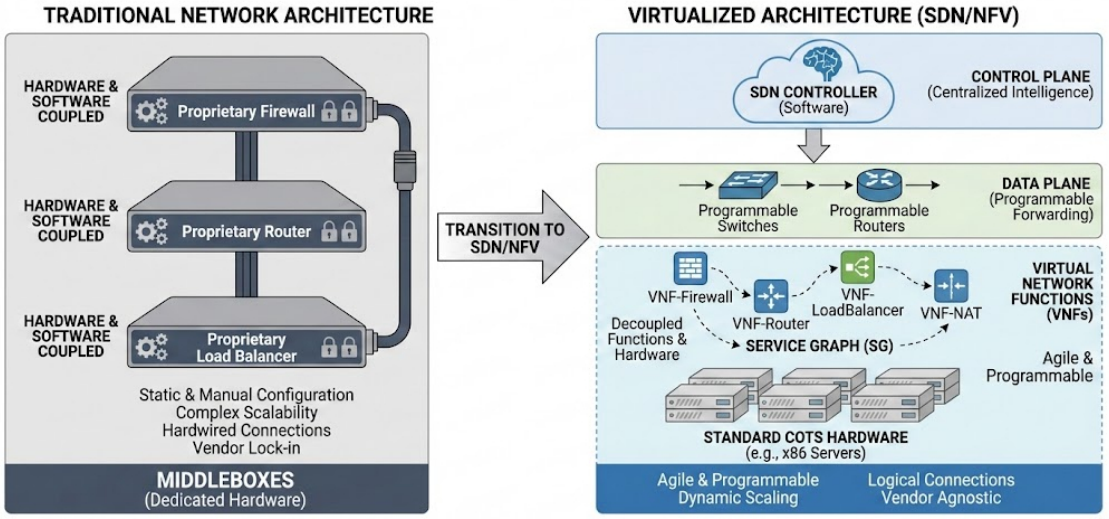


Figure 2.1: Transition from Traditional Architecture to Virtualized Architecture (SDN/NFV)

When combined, SDNs and NFVs enable a high degree of operational agility and flexibility. Complex network services can now be viewed as logical representations of a **Service Graph** (SG) [6, 7] rather than physical chains of network components. A **Service Graph** is a logical representation of a group of connected VNFs (i.e. a client connects to a VNF-Firewall, which then connects to a VNF-LoadBalancer) which can be instantiated, modified, or terminated programmatically in a matter of seconds. While the flexibility provided by this ability to create and modify a **Service Graph** enables automation of security-related tasks [7], it simultaneously creates exponentially greater complexity when compared to managing a single service graph [3].

2.1.2 The Challenge of Security Configuration in Dynamic Environments

While the agile and flexible operation enabled by both SDN and NFV (as defined in Section 2.1.1) enables the creation of more complex services such as **Service Graphs (SG)** [6], it also causes an exponential increase in the amount of effort required to manage the security of those services [7]. In these virtualized environments, the network topology is no longer static; **Virtual Network Functions** (VNF) can be created, moved, or deleted within seconds to adjust to changing workloads or service requirements.

Given the very dynamic nature of these environments, **manually configuring security** becomes largely impractical [3] because traditional methods that rely on human

intervention are subject to two main constraints:

1. **Reaction Time:** Any manual updates to firewall policies or flow reconfiguration in response to changes in the network topology (i.e. scale-up/scale-down of a VNF) will take significant amounts of time. The time spent waiting for the network to react to the changes create "windows" of time during which the network operates in a less-than-secure or less-than-optimal manner [7].

2. **Error Rate:** Distributed policies that are configured manually are inherently *error-prone* [6]. In a distributed and dynamic environment, the lack of a global and automated view of the system will lead to a high likelihood of either incorrect, suboptimal, or non-consistent configurations.

Therefore, the issue is not just to configure something properly, but to provide **formal guarantees of correctness and optimality** [5]. Formal guarantees are required to ensure that the generated configuration will correctly implement all the **Network Security Requirements (NSRs)** (i.e., isolate or allow connectivity between nodes) with the least amount of resources possible (i.e., minimize the number of firewall instances or firewall rules) [7].

Automated Security has become a key concept to solve the issues mentioned above, but simply automating (with scripting) is insufficient since scripting may simply automate the propagation of errors. A formal methodology to provide guarantees of correctness is needed [3].

In this context, the **VEREFOO** (*Verified Refinement and Optimized Orchestration*) framework [6,7], a formal methodology to automatically allocate and configure NSFs (such as firewalls) in virtualized networks and formulate the problem as a MaxSMT instance to produce formally correct and optimal solutions, fits perfectly with the goal of this dissertation.

2.2 Formalized Network and Traffic Flow Model

With the operational environment defined (SDN/NFV) and having identified the need for formal automation due to the limitations of manual configuration (Section 2.1) , it is essential to define the **abstract models** upon which the reasoning of the VEREFOO *framework* is based [3].

For the purpose of using a strict solver-based methodology (MaxSMT) to solve the problem, each aspect of the problem – the network, traffic, and goals/objectives – must be converted into an formal construct.

In this section, the three fundamental models, which represent the *inputs* to the *refinement* and orchestration process, are detailed:

1. **Topology Model** (Section 2.2.1): This section analyzes how VEREFOO translates the network topology provided by the user (**Service Graph** or SG) into the internal topology used for the reasoning process (**Allocation Graph** or AG), which has *placeholders* for position of security functions (*allocation places*).

2. **Traffic Model** (Section 2.2.2): This section outlines how network traffic is represented. The use of **maximal flow**-based modeling [4, 5] will be further explored, a method that aggregates packet flows with the same forwarding and transformation behavior to reduce the complexity of the problem.

3. **Requirements Model** (Section 2.2.3): In this section, the specification of **NSRs** will be formally defined [6] transforming high-level objectives (for example, "isolation" and "reachability") into predicate logic statements that the solver can interpret.

The creation of these three models (topology, traffic, and requirements) is required for the formation of the MaxSMT optimization problem, which is discussed in Section 2.3.

2.2.1 Service Graph (SG) and Allocation Graph (AG)

To address formally the *refinement* issue, it is necessary to establish a rigorous model of the network topology. For this purpose, the *framework* VEREFOO makes use of two levels of abstraction: the **Service Graph** (SG), representing the *input* of the user, and the **Allocation Graph** (AG), representing the internal representation of the solver.

As previously discussed, the fundamental *input* that describes the service topology is the **Service Graph** (SG). This graph represents the logical view of the end-to-end service as described by the *Service Designer* and includes the necessary connections between network functions and nodes to allow the full service delivery. In formal terms, the SG is represented as a directed graph $G_S = (N_S, L_S)$, where L_S is the set of directed edges (connections) among the graph nodes and N_S is the set of graph nodes. The set of graph nodes N_S is defined as the union of two disjoint sets, $N_S = E_S \cup S_S$:

- E_S (*End Points*): It is the set of service terminals, which may be clients, servers or entire subnets that act as entry points for users accessing the services.

- S_S (*Service Functions*): It is the set of network functions (*middleboxes*) needed for the service delivery, such as Network Address Translators (NAT), Load Balancers, Web Caches, etc.

A critical feature of the above representation is that the SG provided by the user is *security-agnostic*, since the SG does not contain security functions (Network Security Functions, NSF) like firewalls, which will be instead identified and allocated automatically by the VEREFOO *framework*.

From the SG, VEREFOO creates internally an additional topological representation called **Allocation Graph** (AG) [3,7]. The AG is the actual graph on which the MaxSMT solver (see Section 2.3) will perform the logical reasoning for the orchestration and configuration of the security functions. The AG has the same structure of the SG, but introduces a new set of nodes. Thus, the set of graph nodes N_A is defined as $N_A = E_A \cup S_A \cup P_A$, where E_A and S_A correspond to the same sets of the Service Graph ($E_A = E_S, S_A = S_S$), whereas P_A corresponds to the set of **Allocation Places**.

Allocation Places (AP) are the basic construct allowing the automatic placement of the security functions. These are *placeholders* introduced by VEREFOO to represent

possible locations for placing an instance of a firewall. In the default generation process of the AG, the *framework* introduces an AP for every edge (*edge*) $l_{ij} \in L_S$ of the original SG. The original edge l_{ij} is substituted by an AP $p_k \in P_A$ and two new edges $l_{ik}, l_{kj} \in L_A$ connecting the original nodes to the AP inserted between them, as shown in Figure 2.2.

Even though the creation of the AG is automatic, VEREFOO provides the *service designer* with the possibility of influencing the placement process through the imposition of certain constraints (*placement constraints*). The designer can:

- **Force allocation** (*forced*): Impose that an AP must host a firewall instance. This possibility is very important to properly model hybrid or existing topologies where some NSFs (such as *hardware appliances*) have been installed and cannot be removed.
- **Forbid allocation** (*forbidden*): Prevent the generation or utilization of an AP on a certain edge. This possibility reduces the size of the search space that the solver needs to explore, thus improving its efficiency at the cost of possible sub-optimality.

In both cases, the constraints ($forced(l_{ij})$ and $forbidden(l_{ij})$) will be transformed in *hard constraints* in the MaxSMT problem, so that the final solution will satisfy the choices made by the designer.

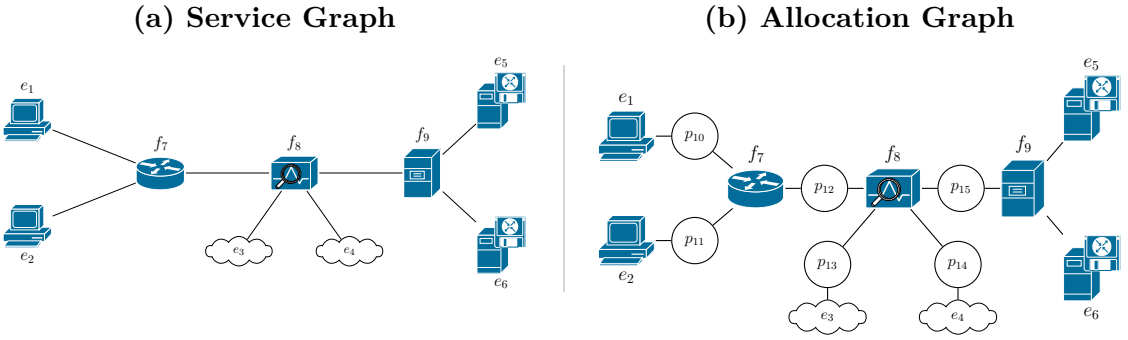


Figure 2.2: Comparison: Service Graph vs Allocation Graph [6]

2.2.2 Traffic and Flow Model (Traffic Flows)

After defining a network's static topology (**Allocation Graph**, Section 2.2.1), the second step in the VEREFOO formal model defines the way traffic flows through that static topology. The VEREFOO *framework* does not examine individual packets, however; it examines the overall traffic classes and how they transform across network paths.

Traffic is the basic unit of analysis for the framework and is called a "packet class" or "traffic." It is formally defined as a predicate on a packet header. More specifically, the model is based on the TCP/IP **5-tuple**: {Source IP, Destination IP, Source Port, Destination Port, Transport Protocol}.

More formally, a packet class $t \in T$ (where T is the set of all possible classes) is a disjunction of predicates $q_{t,i}$ (i.e., $t = q_{t,1} \vee q_{t,2} \vee \cdots \vee q_{t,n}$). Each predicate $q_{t,i}$ is in turn a conjunction of five subpredicates representing the five fields of the 5-tuple:

$$q_{t,i} = (ipSrc \wedge ipDst \wedge portSrc \wedge portDst \wedge trProto)$$

Wild cards (represented with `' '`) may also be used in the model to define a full range (e.g., `' '` for ports $[0, 65535]$) or CIDR notation for IP addresses (e.g., `'10.0.0.*'` for 10.0.0.0/24). The set T is closed under the Boolean logical operators (conjunction, disjunction, and negation) and the symbol t_0 (or *false*) denotes no traffic.

Packet classes in VEREFOO do not depend on topology but are embedded in their end-to-end paths through **Traffic Flows** (F) [3]. A flow $f \in F$ is a formalization of how a packet class generated by a source node n_s is propagated to an ordered list of intermediate nodes (middleboxes and APs) and transformed (e.g., by a NAT) before arriving at the destination node n_d.

A flow f is formally defined as a list of pairs of nodes and traffic classes:

$$f = [n_s, t_{sa}, n_a, t_{ab}, n_b, \ldots, n_k, t_{kd}, n_d]$$

Wherein t_{ij} represents the packet class transmitted from the node n_i to node n_j. The model makes the assumption of **homogeneity**: all packets represented by t_{ij} have to be treated equally by node n_j (all either forwarded/transformed, or all blocked).

Since the number of atomic flows in a given network could be very large, analyzing each flow separately is not feasible. Therefore, VEREFOO uses a key optimization technique called **Maximal Flows** (F^M) [4, 5], which merges into a single representative "maximal" flow all traffic flows having the same behavior (same path and same transformations) and thus creates fewer flows to be analyzed. More formally, the set of maximal flows contains only the flows that are not subflows (*subflows*) of another flow within the same set.

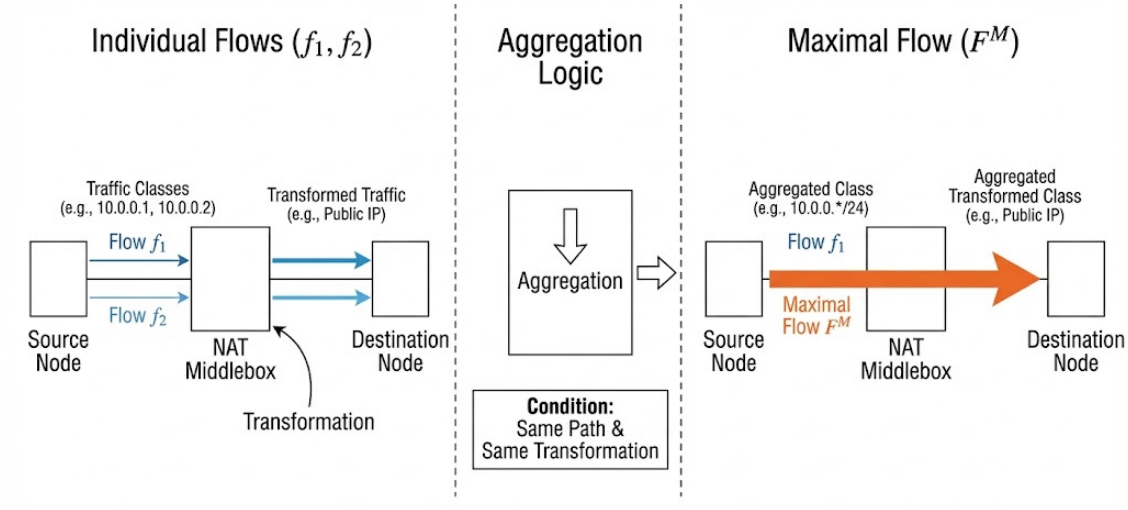


Figure 2.3: Example of aggregating flows (f_1, f_2) into Maximal Flow (F^M)

Using this approach, the cardinality of the set of flows to be analyzed is significantly reduced. Since MaxSMT constraints are created for each identified flow, aggregating them reduces the number of logical clauses the solver has to process, which improves the framework's scalability. Moreover, the MaxSMT problem formulation in VEREFOO (as described in Section 2.4) avoids the use of logical quantifiers (e.g., $\forall n \in N_A$), which

severely reduce the solver's efficiency. The combined usage of pre-computed data structures on the AG (for example `leftHops`/`rightHops` maps specifying the relevant neighbors for a flow) and the calculation of specific paths for Maximal Flows make quantifier elimination possible.

2.2.3 Formalization of Security Requirements (NSRs)

After describing the topological network model (the *Allocation Graph*, Section 2.2.1) and how traffic can travel through it (*Maximal Flows*, Section 2.2.2) - the third building block of the formal model of VEREFOO is the **Network Security Requirements** (NSRs).

NSRs describe security goals of the users (service designers) and are restrictions on network connectivity that the end-network must fulfill. In the case of VEREFOO, an NSR is a specification of which traffic flows should be allowed or forbidden to pass between pairs of end-points. The general form of an NSR r is a pair (C, a), where C is a condition (predicate on the 5-tuple similar to the traffic model) and a is an action; VEREFOO focuses on the two most important actions:

- **Isolation**: $a =$ `DENY`. It requires all traffic flows C to be blocked.
- **Reachability**: $a =$ `ALLOW`. It requires at least one traffic flow C to be allowed to reach the destination.

The key feature of VEREFOO is that each requirement (NSR) is mapped directly to *Hard Clauses* (no relaxations possible) in the MaxSMT problem [6] - thus providing the means of ensuring the formal correctness of the solution (*correctness-by-construction*); by definition, a MaxSMT-solver can only provide a valid solution if all hard constraints are fulfilled.

The formal transformation of these requirements into logical clauses based on *Maximal Flows* (F_r^M, defined in Section 2.2.2) is made necessary by introducing three new auxiliary functions that allow querying the flow-structure:

- $\pi(f)$: maps a flow to the ordered sequence of traversed nodes;
- $\tau(f, n)$: maps a flow and a node to the specific traffic entering that node (taking care of potential transformations);
- $allocated(n)$: predicate indicating whether a firewall is installed on node n.

It is then possible to formally specify the policies:

Isolation Policy ($r.a =$ `DENY`) An isolation requirement demands that for every flow (f) contained in the set of maximal flows of the requirement (F_r^M), at least one firewall on its route blocks the flow. The logical expression is:

$$\forall f \in F_r^M . \exists i . (n_i \in \pi(f) \wedge allocated(n_i) \wedge deny_i(\tau(f, n_i))) \tag{2.1}$$

This expression asserts that for each flow (every $\forall$) that satisfies the requirement, there exists at least one node n_i (some $\exists$) on its route ($\pi(f)$) such that a firewall is

installed ($allocated(n_i)$) and the specific traffic of that flow ($\tau(f, n_i)$) is denied ($deny_i$) by the configuration of that firewall.

Reachability Policy ($r.a =$ `ALLOW`) A reachability requirement asserts that there exists at least one flow $f \in F_r^M$ that is not blocked by any firewall during its entire route. The logical expression is:

$$\exists f \in F_r^M . \forall i . (n_i \in \pi(f) \wedge allocated(n_i) \Rightarrow \neg deny_i(\tau(f, n_i))) \tag{2.2}$$

This expression asserts that there exists at least one flow (some $\exists$) such that for all ($\forall$) nodes n_i on its route, if a firewall is installed ($allocated(n_i)$), it follows ($\Rightarrow$) that its configuration does not deny ($\neg deny_i$) the traffic of that flow.

These constraints will serve as the foundation for correctness verifications and, as shown in Section 2.5, they will be further enriched with the specific semantics of stateful firewalls.

2.3 Input Specification: XML Schema and Processing

In this section we explain how the designer interacts with VEREFOO, and therefore why the logical architecture and formal models (the *Allocation Graph*, *Maximal Flows*, and *Network Security Requirements* (NSRs)) described in Section 2.2 are abstracted internally as a mathematical representation of the framework to apply formal methods.

To allow designers to enter information about the design problem they wish to solve, it is essential that designers have an interface with VEREFOO. Therefore, a method to specify input information must exist. The input specification mechanism employed by the framework is an XML (*eXtensible Markup Language*) schema [3].

XML was selected for the input specification mechanism because it is able to accurately represent the complex network topologies and policy structures in a hierarchical and structured form.

This Chapter explains the detailed structure of this input. Section 2.3.1 explains the XML schema used to define the *Service Graph* topology (nodes, node functions, etc.), Section 2.3.2 defines the format in which NSRs are entered, and Section 2.3.3 describes the parsing mechanism that transforms the XML input files into logical representations and formal models (from Section 2.2) that will then be used to formulate a MaxSMT problem (the subject of Section 2.4).

2.3.1 XML Structure for Service Graph Definition

To allow the parser to read the topological input that is required by the VEREFOO *framework*, the designer of the service defines the network in a structured XML format. The concrete form of this structured XML format transforms the abstract Service Graph model ($G_S = (N_S, L_S)$) of Section 2.2.1 into a format that may be processed by the *framework*'s *parser*.

All topological information resides in a single root element `<graphs>`. In addition to the root element `<graphs>`, zero or more `<graph>` elements exist. All `<graph>` elements are assigned a unique `id` as an attribute and are assigned the attribute `serviceGraph="true"`.

This distinction is necessary because a similar allocation graph exists internally (as described in Section 2.2.1) but has no standardized way for users to define.

Each `<node>` element represents a vertex $n \in N_S$ (either an *End Point* E_S or a *Service Function* S_S). The basic attributes of this element are:

- `name`: The name of the node; typically the node's IP address (e.g., "10.0.0.1") or a subnet identifier (e.g., "10.0.0.*"); serves as a primary key for the node.
- `functional_type`: The attribute describes the semantics of the node; is important for the parser to determine if a given node is an End Point (e.g., `WEBCLIENT`, `WEBSERVER`) or a Service Function (e.g., `NAT`, `CACHE`, `TRAFFIC_MONITOR`).

Edges in the graph (L_S) do not have individual definitions as standalone elements; rather, they are implied through the use of nesting. Within each `<node>` element are listed one or more `<neighbour>` elements, each specifying the name of an adjacent node; thus defining a directed edge from the parent node (`<node>`) to the child node (`<neighbour>`).

Nodes representing Service Functions (S_S), which require special configuration to perform their functions (e.g., a NAT needs to know the source addresses in order to properly translate them), contain a `<configuration>` block. The block contains function-specific elements (e.g., `<nat>`, `<cache>`) describing how the Service Function behaves.

An example of a simple Service Graph, shown below, demonstrates the XML structure.

```
<graphs>
  <graph id="0" serviceGraph="true">

    <node functional_type="WEBCLIENT" name="10.0.0.1">
      <neighbour name="20.0.0.3"/>
      <configuration ... >
        <webclient nameWebServer="30.0.5.2"/>
      </configuration>
    </node>

    <node functional_type="NAT" name="20.0.0.2">
      <neighbour name="20.0.0.3"/>
      <neighbour name="20.0.0.1"/>
      <configuration ... >
        <nat>
          <source>10.0.0.1</source>
          <source>10.0.0.2</source>
        </nat>
      </configuration>
    </node>

  </graph>
</graphs>
```

2.3.2 Specification of Network Security Requirements

In addition to the Service Graph topology (described in Section 2.3.1), the XML input file has to include the definition of the **Network Security Requirements (NSRs)** which have to be satisfied by the final configuration. The XML elements translating the formal requirements model (defined in Section 2.2.3) into a parser-readable format are called Requirements.

Each Requirement is contained inside an element `<PropertyDefinition>`. Elements of type `<PropertyDefinition>`, group one or more `<Property>` elements, each of them specifying a single security requirement (NSR). Each element corresponds directly to the fields of the 5-tuple and the desired action as follows:

- `name`: specifies the desired action a (defined in Section 2.2.3). It corresponds to the `ReachabilityProperty` for a reachability requirement ($a =$ `ALLOW`) or to the `IsolationProperty` for an isolation requirement ($a =$ `DENY`).
- `src` / `dst`: map fields $C.IPSrc$ and $C.IPDst$. They define the source and destination IP addresses.
- `src_port` / `dst_port`: map fields $C.pSrc$ and $C.pDst$. They define the source and destination transport port.
- `l4proto`: map field $C.tPrt$. It defines the transport protocol (`TCP`, `UDP`, etc.) or `ANY`.
- `graph`: a management attribute giving the id of the graph (defined in Section 2.3.1) on which this requirement is applied.

To allow a flexible input, the VEREFOO XML schema allows the use of wildcards and intervals (intervals) in the values of these attributes. For IP addresses, the framework can use `-1` as a wildcard in an octet (e.g., `dst="130.192.-1.-1"` for the subnet 130.192.0.0/16). For ports, it is possible to use `*` as a wildcard (all ports) or an explicit interval (e.g., `src_port="[2000-3000]"`).

The following example shows how the XML definition of a set of NSRs could look like:

```
<PropertyDefinition>

  <Property graph="0" name="ReachabilityProperty"
            src="10.0.0.1"
            dst="20.0.0.2"
            dst_port="80"
            l4proto="TCP"/>

  <Property graph="0" name="IsolationProperty"
            src="10.1.1.1"
            dst="130.192.-1.-1"
            src_port="[2000-3000]"
```

```
            dst_port="80"
            l4proto="ANY"/>

</PropertyDefinition>
```

This elements are parsed by the parser (Section 2.3.3) and they are then used to compute the maximal flows (Section 2.2.2) and hard constraints (Section 2.2.3) needed to build the MaxSMT problem (Section 2.4).

2.3.3 From XML to Formal Models

The parsing process represents the main connection between the user's concrete description given in the form of XML files (Sections 2.3.1 and 2.3.2), and the internal mathematical abstraction needed by the solver (the formal models of Section 2.2). This phase is managed by specific software components inside the VEREFOO system, specifically the `VerefooSerializer`, which act as translators.

Firstly, the parsing process starts with parsing the XML input file. The parser analyzes the XML document's structure to create Java objects representing the network topology. The parser maps each element to an `AllocationNode` object, and the elements are used to construct the graph structure (the *Allocation Graph*). At the same time, the parser reads the elements and maps them to `SecurityRequirement` objects.

After the XML input has been converted into its object representation using the above-described process, the framework proceeds with translating the object representation to the formal model. This phase lays the foundation for MaxSMT resolution:

- **Flow Generation:** Once the topological and requirement data (Network Security Requirements or NSRs) have been analyzed by the framework, it calculates the set of *Maximal Flows* based on the aggregation logic explained in Section 2.2.2.
- **Constraint Construction:** Based on the flow values calculated in the previous step, the VEREFOO framework constructs the logical problem instance. In this phase:
  - NSRs are converted into *Hard Constraints*, i.e., mandatory logical clauses ensuring network security (the details of which will be covered in Section 2.4.1).
  - User preferences and efficiency metrics are translated into *Soft Constraints*, optional weighted constraints providing guidance for optimization (further detailed in Section 2.4.2).

Ultimately, this process results in the creation of a complete formal logical model that fully abstracts from the original XML input. The z3/z3Opt engine then uses this mathematical model for solving, as further detailed in the next Section.

2.4 The MaxSMT Solution: Optimality and Constraint Structure

Although VEREFOO has defined the network architecture (Allocation Graph), traffic behavior (Maximal Flows), and security requirements (NSRs), it still remains as an abstract problem from the algorithmic point of view; that is, which is the correct way to configure the firewalls so that they comply with all the previous constraints.

In terms of solving this type of problems using *Satisfiability Modulo Theories* (SMT) it would be sufficient to obtain any valid configuration that respects the security requirements. However, since we are working with virtualized networks (NFV), the correctness of a solution is no longer sufficient. In fact, it is necessary to find an **optimal** solution concerning resources usage, i.e., minimizing the number of allocated network functions, the number of configured rules, and the impact on network performance. A SMT Solver does not support such an objective, as it will stop when the first valid solution is found, even if the solution is over-dimensioned.

Therefore, in order to reconcile both the rigidity of security requirements and the flexibility of resource optimizations, VEREFOO models the orchestration problem as a **Weighted Partial MaxSMT** problem [6], that extends the SMT Logic by distinguishing between two hierarchical classes of constraints:

- **Hard Clauses** (Rigid Constraints). They represent conditions that have to be obligatorily fulfilled by the solution in order to be considered valid (e.g. security requirements).
- **Soft Clauses** (Relaxable Constraints). They represent desirable conditions but not mandatory ones (e.g.: "do not use a firewall here"). A cost (a weight) is assigned to each violation of a soft constraint.

The objective of the solver is then to find a variable assignment that fulfills all Hard Clauses and, in addition, minimize the sum of the weights of violations of Soft Clauses.

The next sections describe how both components are mathematically formulated in order to provide guidance for the refinement process.

2.4.1 Formulation of the Problem as MaxSMT

VEREFOO must convert the network problem into a math problem, which means defining what "knobs" the solver may use to solve the problem. Those are the decision variables. The system must provide answers to two very basic questions about each of the points in the network: "Do I put a firewall here?" and "If I do, which kind of packets should I block?"

Decision variables are represented with two types of Boolean variables (which have just one of two states: *True* or *False*):

- **Allocation Variables** ($allocated(p_k)$): There is one allocation variable for every potential place (p_k) in the graph. This variable determines whether to set up a firewall at that place, if it is *True* then a firewall is placed there.

- **Configuration Variables** ($deny(n,t)$): If a firewall is in place on the node (n), this variable determines whether to enable or disable a certain blocking rule for a particular traffic flow (t). If it is *True*, the specified type of traffic will be blocked, otherwise it will not.

Variables representing the allocation and configuration variables cannot have arbitrary values but must comply with hard logic rules. There are two groups of these rules: **structural coherence** and **actual security**.

Structural coherence is the most obvious example of a hard constraint: you cannot establish a blocking rule on a non-existent node. This is expressed in the form of an implication in mathematics:

$$deny(n,t) \implies allocated(n)$$

As mentioned before, the second group of hard constraints represents real-world security. The solver will search through all of the possible combinations of switches (*allocated* and *deny*) and identify the one that satisfies both the isolation and reachability criteria provided by the user (see Section 2.2.3). In other words, the solver searches through millions of possible combinations until it finds one that meets the following conditions:

1. All required firewalls are turned on.
2. All necessary blocking rules are enabled.
3. No traffic that was prohibited from passing through can pass through, and all traffic that was permitted to reach its intended destination will reach that destination.

In case such a combination is found, the problem is solvable (SAT). Once the problem is identified as being solvable, the cheapest combination of switches that satisfy the problem will be selected (this is discussed further in the next section on soft constraints).

2.4.2 Relaxed Constraints (Soft Constraints) - for Optimizing Resource Consumption

Following the establishment of the Correctness of Configuration through *Hard Constraints*, VEREFOO's second objective is to optimize resources (in an NFV context) to minimize network usage; specifically to reduce the Number of Instantiated Firewalls and the Complexity of Internal Firewall Rule Sets.

Optimization is achieved through the use of *Soft Constraints*, which are different from rigid constraints. Instead of imposing strict obligations on the Solver, they identify a preferred (or default) configuration State that the Solver attempts to maintain. When the Solver can no longer maintain the preferred State due to a Security Requirement, a Penalty (Weight) is added to the Global Objective Function.

For VEREFOO, Optimization is modeled using two primary Negative Soft Constraints [7]:

1. **Minimizing Resource Allocation**: This constraint is expressed as follows: "at this moment (p_k) there is no firewall"

$$Soft(allocated(p_k) = false, W_{fw})$$

When the Solver manages to maintain $allocated(p_k)$ false, the Cost is Zero. When the Solver must assign a Firewall ($allocating(p_k) = true$) to meet a Security Requirement, the Total Solution Cost is increased by the Weight W_{fw}.

2. **Minimizing Number of Rules**: Similarly, for each possible Rule, the System expresses the constraint: "the blocking rule is not configured."

$$Soft(deny(n, t) = false, W_{rule})$$

Again, when a Rule is activated ($deny = true$), the Solver adds the Weight W_{rule} to the Cost Function.

Thus, the Optimization Strategy is simply a Weighted Sum. The Solver determines the "Total Cost" of each Valid Solution using a Linear Function:

$$Cost_{total} = (N_{firewall} \times W_{fw}) + (N_{rules} \times W_{rule})$$

Where N represents the Number of Active Elements, and W represents their Unit Cost (i.e., Weight).

By adjusting these two Weights, we can express the System's Priority. For instance, if we set $W_{fw} = 1000$ and $W_{rule} = 10$, the Solver is being told that a new Firewall "costs" as much as 100 New Rules. Thus, the mathematical Algorithm would naturally prefer to add more Rules to a currently Existing Firewall, rather than create a new Firewall to fill it, thus maximizing Resource Consolidation.

2.5 VEREFOO's Extension for Stateful Firewalls

Previous sections (2.2 and 2.3) have described the Formal Approach of VEREFOO to Firewall Refinement and Optimization. It was demonstrated how Topology (AG), Traffic (*Maximal Flows*) and Requirements (NSRs) were converted into a MaxSMT Problem. However, while Section 2.4.2 has presented the Logic of Optimization for Stateful Firewalls (preference for `ALLOW_COND`), it has still not provided the Formal Semantics of this extension. While the Base Model (`ALLOW/DENY`) remains fundamentally *Stateless* and does not capture Connection Memory Management, a key Feature of Stateful Firewalls examined in Chapter 1.

Therefore, the limitation of the base model needs to be addressed to allow the Solver to orchestrate and configure *Stateful Firewalls* that not only Filter Packets according to the 5-Tuple, but also Track the Connection State (e.g., `NEW`, `ESTABLISHED`, `RELATED`).

This Section describes the Architectural and Logical Modifications made to the Framework. Section 2.5.1 will detail the Integration of Stateful Semantics into the Model, Defining New Categories of Traffic (I_i^{as}, I_i^{ac}) and Introducing the new `ALLOW_COND` Action. Subsequently, Section 2.5.2 will Examine the Formalization of this Logic, Presenting the

New Hard Constraints (such as the `cond_permit` predicate) Necessary to Correctly Model Conditional Permission and Return Traffic [14].

2.5.1 Integration of Stateful Semantics

The need to go beyond the VEREFOO framework's classical traffic model to support stateful firewalls required overcoming its limitations. As previously presented in literature [7] the classical model is inherently stateless. It defines each incoming packet to a node n_i as being either an allowed packet (I_i^a) or a denied packet (I_i^d). Given that the filtering decision depends on the history of the flow and the temporal aspect of the flow [14], a more granular representation of the traffic space is needed to model the operational decisions taken by the Connection Tracker. Thus, the new traffic model distinguishes the incoming traffic to each node n_i into four classes of traffic [4]:

- I_i^a (*Accept*): The class of packets accepted by the firewall (i.e., stateless behavior), with no record created in the connection table.
- I_i^d (*Drop*): The class of packets that are always blocked by the firewall, regardless of any prior connection state.
- I_i^{as} (*Accept and Save State*): The class of packets accepted by the firewall, for which a new entry is created in the State Table. These correspond to the first packets of a new connection initialization (e.g., `TCP SYN` or the first UDP packet) and are classified as `NEW` state.
- I_i^{ac} (*Accept Conditionally*): The class of packets accepted by the firewall only if a valid corresponding state already exists in the table. These correspond to return or subsequent packets (e.g., `TCP ACK`) and are classified as `ESTABLISHED` or `RELATED` state.

This new classification enables us to formally describe the relationship between outgoing traffic (which generates state) and return traffic (which consumes state). Classes I_i^{as} and I_i^{ac} are inversely related. Formally, we can express the conditional traffic class I_i^{ac} as the disjunction of the inverse predicates of those that generate state:

$$I_i^{ac} = \bigvee_x \text{inv}(q_{x,i,as}) \tag{2.3}$$

where:

- $q_{x,i,as}$ is an atomic predicate of the set of state-accepted flows (I_i^{as});
- $\text{inv}(q)$ is a function that reverses the directional components of the 5-tuple of predicate q (exchanging source IP address with destination IP address and exchanging source port number with destination port number).

The formula (2.3) represents the idea that the firewall will "conditionally" accept only those packets that represent the direct response to a previously authorized and stored request [14].

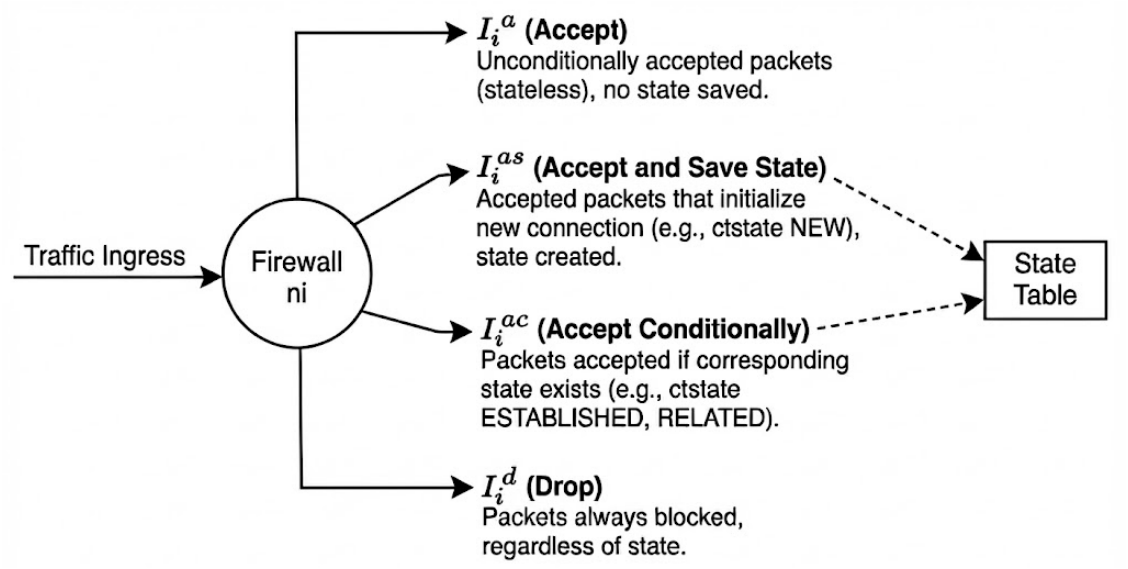


Figure 2.4: Traffic model classification for stateful firewalls [14].

To enable the MaxSMT solver to use this new traffic model, VEREFOO provides a new refinement action called `ALLOW_COND` (*Conditional Permission*), which abstracts the common bidirectional logic of a stateful rule into a single logical construct:

- An `ALLOW` action (stateless) grants permission to flow only in one direction (A $\rightarrow$ B), without memory.
- An `ALLOW_COND` action on a flow A $\rightarrow$ B means two simultaneous semantical aspects exist:
  1. Unconditional permission to forward traffic A $\rightarrow$ B (thus populating set I_i^{as}).
  2. Conditional permission to return traffic B $\rightarrow$ A (thanks to the inverse relation defined above; thus populating set I_i^{ac}).

Enabling stateful behavior in the traffic model makes possible the characterization of the security policies, and how they relate to different types of network connectivity requirements that the framework needs to fulfill. We have identified four major categories of Network Security Policies (NSPs) in the VEREFOO context [14]:

- **Isolation Policy**: States that for all flows fulfilling the requirement conditions, at least one security function must be present along the path that is configured to block the traffic. This definition is independent from the connection state: the traffic cannot reach the destination (i.e., `DENY`).
- **Simple Reachability Policy**: States that a valid flow must exist in such a way that traffic is not blocked along the path. In a stateful environment, this policy is fulfilled when the firewall grants permission to packet transmission, regardless of the fact that the decision is based on a static rule or the existence of a prior state.

- **Strong Reachability Policy**: States that the traffic must be granted unconditional access, i.e., independently of the existence of a prior state. This policy is generally applied to model the connection initialization flow (forward traffic, e.g., `SYN` packets), which must be explicitly authorized through a static rule to create an entry in the State Table (class I_i^{as}).

- **Conditional Reachability Policy**: States that the traffic is allowed to pass only if a valid prior-established connection state exists. This policy models return traffic (backward traffic, e.g., `ACK` packets or response data), which does not require explicit static rules to be authorized dynamically due to the Connection Tracking mechanism (class I_i^{ac}).

With the definitions of the new traffic classes and the new `ALLOW_COND` action, it is now necessary to formalize the constraints (*Hard Constraints*) that govern their behavior in the z3 solver. This will be addressed in the next section, which will introduce the logical predicate `cond_permit`.

2.5.2 Formalization of Conditional Permission

Due to the introduction of new stateful categories (Section 2.5.1), especially I_i^{as} and I_i^{ac}, the formal constraints (*Hard Constraints*) applied by the MaxSMT solver must be directly extended. The Stateless Model (Section 2.2.3) provides conditions only for I_i^{a} (Must Permitted) and I_i^{d} (Must Blocked). The formalization of behaviors for I_i^{as} (Accept and Save State) and I_i^{ac} (Accept Conditionally) is required.

For the I_i^{as} class, the constraint is simply: packets which establish a new state must be permitted. The constraint for I_i^{as} packets is formally identical to the constraint for I_i^{a} packets:

$$\forall q \in I_i^{as}.\forall p \in q.\neg deny_i(p)$$

The technical and conceptual difficulty is formalizing the I_i^{ac} class. Such packets can only be permitted when a valid state exists, i.e., when the reverse traffic ($inv(q)$) has been processed by the preceding firewall and classified as I_i^{as}. In order to represent the conditional logic without having a runtime state table, VEREFOO introduces a new Boolean predicate, `cond_permit(n, t)`.

This predicate is true if and only if traffic t, arriving at node n, is of type I_i^{ac}. In other words, the predicate `cond_permit` identifies traffic that cannot be blocked, but must have been accepted on the basis of a previously established state.

Thus, the conditional permission logic is represented by the fourth class of constraints (a new *Hard Constraint*) relating the $deny_i$ decision to the existence of a valid reverse flow. A packet p (which is understood as the header that uniquely identifies the traffic unit) of type $q \in I_i^{ac}$ (i.e., $cond\_permit_i(p)$ is true) is not denied ($\neg deny_i(p)$), if and only if ($\Leftrightarrow$) the state condition (equation 2.4) is met:

$$\forall q \in I_i^{ac}.\forall p \in q.(\neg deny_i(p) \Leftrightarrow \otimes)$$

The condition $\otimes$ (equation 2.4) represents the verification of the state. It is true if and only if the inverse traffic $p' = inv(q)$ can be associated with some existing flow f in the set F_R (the set of all the traffic flows modeled in the system, or *Maximal Flows*, corresponding to a given requirement), satisfying simultaneously three conditions:

$$\otimes = \forall p' \in inv(q). \exists f \in F_R. (1) \wedge (2) \wedge (3) \tag{2.4}$$

Where:

1. $\tau(n_i, f) = p'$: The reverse traffic p' (corresponding to packet p) actually arrives at node n_i as part of flow f.

2. $\tau(n_i, f) \wedge I_i^{as}$: That reverse traffic p' has been classified by firewall n_i as I_i^{as}, i.e., it has been authorized to create state.

3. $\forall n_j \in \pi(f) | n_j < n_i. \neg deny_j(\tau(n_j, f))$: No other node n_j, located before firewall n_i along the path of reverse flow f, has blocked such flow.

This formulation shifts *connection tracking* logic from a *runtime* mechanism (the *state table* analyzed in Chapter 1) to a formal logical constraint defined a priori (*compile-time*) in the MaxSMT problem.

2.6 Implications and Benefits of Using VEREFOO

Having described VEREFOO's formal approach for describing distributed systems (from topology modeling to the MaxSMT problem and refinement as MaxSMT problem) in the previous chapters (2.2, 2.3, and 2.4), we can now examine the theoretical and operational implications and benefits associated with the use of this methodology.

There are two main implications arising from the adoption of a formal refinement process (by means of MaxSMT) to generate secure configurations that are *correct-by-construction* instead of relying on *runtime* state management (which was analyzed in Chapter 1):

In Section 2.6.1, we will show the benefits of using VEREFOO's formal approach to manage distributed security. We will specifically describe how VEREFOO defines a globally correct configuration during the orchestration phase, thereby avoiding the difficult task of synchronizing states (*state synchronization*) among multiple components in the same system or across multiple systems in resilient or virtualized environments, which is the primary weakness of stateful firewalls in those architectures.

In Section 2.6.2, we will analyze theoretically and experimentally the efficiency and scalability of the VEREFOO approach. While MaxSMT problem solving is a well-known NP-complete problem, we will explain how optimization methods (such as logical quantifier elimination and pruning) included in our models make it computationally viable (and therefore scalable) for very large-scale scenarios, paving the way for the experimental validation that constitutes the central part of this dissertation.

2.6.1 Benefits in Distributed Security

VEREFOO's formal approach provides a crucial benefit when managing distributed security in networks. Traditionally, *runtime* state synchronization issues have been resolved at *runtime*; however, VEREFOO resolves them at *design time*.

When there are asymmetrical routes (forward and reverse flows travel through different nodes) and stateful firewalls are used, the connection tracking tables of stateful firewalls must be synchronized in real-time in order to allow return traffic. Without state synchronization, a firewall on a return path n_B may allow return traffic based on weak rules, exposing the network to *spoofing* attacks (e.g., fake `SYN-ACK` or `ACK` packets).

Our model prevents this risk *by construction* since the acceptance of return traffic is strictly bound to the entire flow lifecycle and therefore is subject to a hard logical constraint for conditional acceptance (I^{ac}).

$$\forall q \in I_i^{ac}, \forall p \in q. \neg deny_i(p) \Leftrightarrow \otimes$$

The composite condition $\otimes$ (Equation 2.4) statically verifies the flow's history. Specifically, the logical symmetry clause:

$$\tau(n_i, f) = p'$$

states that, for node n_i to instantiate a conditional permission rule for the return packet p, it must have also processed the corresponding forward packet p' (where $p' \in inv(p)$) and classified it as I^{as} (*Accept and Save*).

Resolution Example. For example, consider a network where a flow f traverses firewall n_A in the forward direction (creates the state) and firewall n_B in the return direction. If the model attempts to define a conditional permission rule to allow n_B to accept the return traffic ($p \in I_B^{ac}$), the verification of the logical symmetry clause fails because $\tau(n_B, f) = \emptyset$ (node n_B did not observe the forward traffic p'). Therefore, the constraint $\otimes$ evaluates to *false*, making the creation of the permission rule on n_B unsatisfiable (`UNSAT`).

Therefore, the MaxSMT solver is forced to reject any insecure configurations, and therefore converge toward solutions that satisfy flow symmetry or the placement of firewalls at topologically symmetric points, eliminating *spoofing* vulnerabilities at their origin, without requiring state synchronization at *runtime*.

2.6.2 Efficiency and Theoretical Scalability

The second important implication of VEREFOO relates to its efficiency and scalability, although theoretically the chosen methodological approach has several drawbacks. Formulating the refinement problem as a MaxSMT instance (Section 2.4.1) is, in terms of computational complexity, equivalent to the class of NP-complete problems. Thus, it is possible that the time required to find the optimal solution could increase exponentially with an increasing number of instances (number of Allocation Places, P_A, and number of Requirements, R).

However, VEREFOO is not designed to solve the worst-case scenarios, but rather to provide a practically efficient solution for network scenarios. To achieve this goal,

VEREFOO uses two fundamental optimization methods to dramatically decrease the size of the search space of the z3 solver [7]:

- **Reduction of Soft Constraints:** As explained in Section 2.4.2, the time required for the resolution is mainly due to the number of soft constraints, which directly depends on the number of placeholder rules (Π_k) generated for each firewall. VEREFOO employs reduction strategies to aggressively eliminate unnecessary rules (e.g., requirements that are satisfied by the default `DENY` action in whitelisting mode, or requirements whose flows do not actually pass through that AP).
- **Elimination of Logical Quantifiers:** The most effective optimization strategy used in the model is the complete removal of logical quantifiers (e.g., $\forall n \in N_A$, "for all nodes") in forwarding formulas. Previous studies and research work employing quantifiers required the solver to expand the formula for all possible pairs of nodes in the network, regardless of whether they represent a valid path, leading to an explosive growth of the search space. VEREFOO solves this limitation by introducing `leftHops` and `rightHops` data structures that are pre-calculated on the Allocation Graph [3]. Each node in the graph is uniquely associated with valid adjacent nodes for a given flow through `leftHops` and `rightHops`: `leftHops` indicates the preceding nodes from which a packet can be received legally, while `rightHops` indicates the succeeding nodes to which it can be forwarded legally. Due to these structures, MaxSMT formulas are expressed in terms of punctual constraints between specific nodes (e.g., "if I receive from A, then forward to B"), and therefore, the need to evaluate the entire node domain is eliminated, significantly reducing the computational cost of the problem presented to the solver.

Through the combination of these two strategies, the theoretical NP-complete complexity of the MaxSMT problem is addressed, ensuring that the VEREFOO framework scales correctly. As will be shown in the experimental validation (Chapters 5 and 6), this approach proves to be efficient and reliable even in complex and large-scale network environments, such as Texas2000.

Chapter 3

Objective of Thesis

The chapters above outlined the technical environment (Chapter 1) and the formal structure of the VEREFOO framework (Chapter 2). Although MaxSMT methodologically ensures *correctness-by-construction* [7], the actual feasibility of the application is dependent on the quality of the implementation and the solvability of the problem within reasonable time frames.

In this chapter we will outline the goals of the thesis project, specifically focusing on the experimental verification of two important features: the logical validity of new stateful functionalities and the scalability of the framework under realistic conditions.

3.1 Background & Objectives

VEREFOO's methodology is based on translating high level security requirements into logically very complex constraints [6]. The addition of stateful semantics, especially with regard to conditionally permitting actions (`cond_permit`) and return flows, increases the complexity of the model exponentially. However, despite the fact that there are theoretical soundness arguments, practical implementation in an SMT solver such as Z3 [14] brings concrete risks:

- **Risk of Logical Inconsistency**: Errors during translation of formulas and/or errors in the management of dependency flows could result in the solver generating configurations that, although mathematically correct, do not meet the original security requirements.
- **Risk of Computation Explosion**: Due to the NP-completeness of the MaxSMT problem and due to the stateful constraints, the framework may become unusable on realistically sized networks.

Therefore, the goal of this thesis is not the theoretical expansion of the framework, but the empirical verification of the framework via two main paths:

- **Objective 1 (Correctness)**: Show that the logical implementation of stateful policies is correct and free from ambiguity.

- **Objective 2 (Scalability)**: Prove that optimization methods allow the applicability of the framework to be extended to large-scale scenarios.

3.2 Objective 1: Verification of Logical Correctness

The first objective is to verify that the configurations produced by VEREFOO satisfy the Network Security Requirements (NSRs) when using stateful firewalls. It is necessary to guarantee that the `ALLOW_COND` action and NAT management do not create vulnerabilities or unwanted blocks.

Cross-Validation is the methodology used to achieve this objective. The framework is run in two different and sequential ways:

1. **Refinement Mode**: The generation of the optimal configuration from the requirements.
2. **Verification Mode**: Formal static verification of the generated configuration with respect to the same requirements.

Verification is successful if and only if both phases confirm the satisfiability (SAT) of the constraints. The details of the test suite implementation, including the specific examples of NAT and asymmetric flows, are described in Chapter 4.

3.3 Objective 2: Validation of Scalability & Performance

An otherwise correct formal solution is useless if it takes infinitely long to compute. Therefore, the second objective is to evaluate the operational limits of the framework, specifically measuring the scalability of the execution time and the memory usage as a function of increasing network complexity.

To provide results relevant to real-world applications (e.g., critical infrastructure), simply using random topologies is insufficient. Therefore, the validation utilizes a specific case study: the **Texas 2000** model [9, 13]. The **Texas 2000** model represents the Texas power grid in a synthetic but realistic way in order to support large-scale research without revealing sensitive information. This model provides the hierarchical and dimensional complexity required to put pressure on the resolution engine. The description of the test generator and the topology are presented in Chapter 5; the experimental results are analyzed in Chapter 6.

Chapter 4

Correctness Validation

In Chapter 3, we identified the two primary goals of this research project: Cross-Validation of Correctness (Objective 1) and Performance Validation (Objective 2). In this chapter, we are solely concerned with validating Correctness (Objective 1), as defined in Chapter 3.

As stated in Section 3.1, the Formal Approach of VEREFOO, using MaxSMT, guarantees Correctness-by-Construction [7]. Nonetheless, due to the complexity of implementing stateful semantics (Section 2.5), specifically the handling of return flows (`ALLOW_COND`) and interactions with packet-altering NFs (like NATs), experimental verification is required to confirm there are no Logical Errors or Implementation Errors [14]. As previously mentioned in Section 3.2, our aim in this chapter is to prove that the stateful firewall configurations created through VEREFOO's refinement process correctly implement the original NSRs.

To verify the previous statement, the chapter will follow the cross-validation methodology described in Section 3.2. Section 4.1 will give a brief overview of the methodological workflow and software components (test runner, serializer, z3 solver) necessary to perform the cross-validation in two phases (*Refinement* and *Verification*).

The bulk of the validation will be represented through the test case designs. Section 4.2 will outline the test case selection criteria and motivate why certain scenarios were chosen to "stress" the most difficult logic within the framework; namely: Interaction with NATs (correctness of state inversion), Management of Multiple Concurrent Flows (control of state interference), and Multi-Firewall Topology (consistency without runtime synchronization) [3].

Section 4.3 will document an inventory of test files utilized (Test Artifacts Inventory); Section 4.4 will illustrate how the test files represent representative examples of the tested logic, including the input data, tested logic and the expected output to validate correctness for example with regard to Port/Wildcard management.

Lastly, Section 4.5 will define the key metrics for validation (primarily the consistency of results – SAT/UNSAT – between the two phases) and will summarize the results of the validation, confirming the logical correctness of the stateful model and justify moving forward to assess performance (Objective 2).

4.1 Methodological Overview

To validate the logical correctness (Objective 1, Section 3.2) of the stateful firewall configurations produced by VEREFOO as being consistent with the specified Network Security Requirements (NSRs), we employed a rigorous cross-validation methodology, utilizing the framework in two distinct operational modes: First as a refinement engine and then as a verification engine, thus providing a cross-check on the outputs [6].

The test case execution workflow can be described as follows:

Phase 1: Refinement

- **Inputs**: The test case is initiated with an XML input file obtained from the `StatefulCorrectness` test suite. The file represents the Service Graph (SG) topology, network functions and a set of NSRs.
- **Execution**: VEREFOO executes in refinement mode. The framework converts the inputs into formal representations and formulates the problem as a MaxSMT instance and invokes the z3Opt solver [7].
- **Output**: If the problem is solvable (SAT), z3Opt generates an optimal solution consisting of firewall allocation and a set of concrete filtering rules.

Phase 2: Verification

- **Input**: This phase utilizes the same original topology and NSRs as input, plus the firewall configuration generated by Phase 1.
- **Execution**: A second instance of VEREFOO is invoked in verification mode. In this mode, the framework seeks to verify that the given configuration satisfies all requirements by formulating an SMT problem for each one [3].
- **Output**: The output of this phase is a Boolean value (SAT/UNSAT) for each requirement, indicating whether the generated configuration satisfies the requirement or not.

The *Test Runner* is a dedicated Java class, `TestStatefulFirewallPolicy.java`, which orchestrates the entire cross-validation process. At an implementation level, the *Test Runner* facilitates communication between the two phases using the file system. When the refinement phase has produced a SAT result, the resultant configuration (an instance of the *Allocation Graph*) is serialized by the `VerefooSerializer` into a new temporary XML file. Within this file, the graph attribute is set to `serviceGraph="false"` to signal that it is a static configuration to be verified. Following serialization, the *Test Runner* invokes the verification phase upon this temporary file. After the verification phase completes, the temporary file is removed.

A successful validation for a test case is defined by the **Consistency of Results**: If the refinement phase produces a valid configuration (SAT result), the verification phase should confirm that the valid configuration satisfies all NSRs. Any inconsistencies could indicate an Error in the Formal Logic of the Framework.

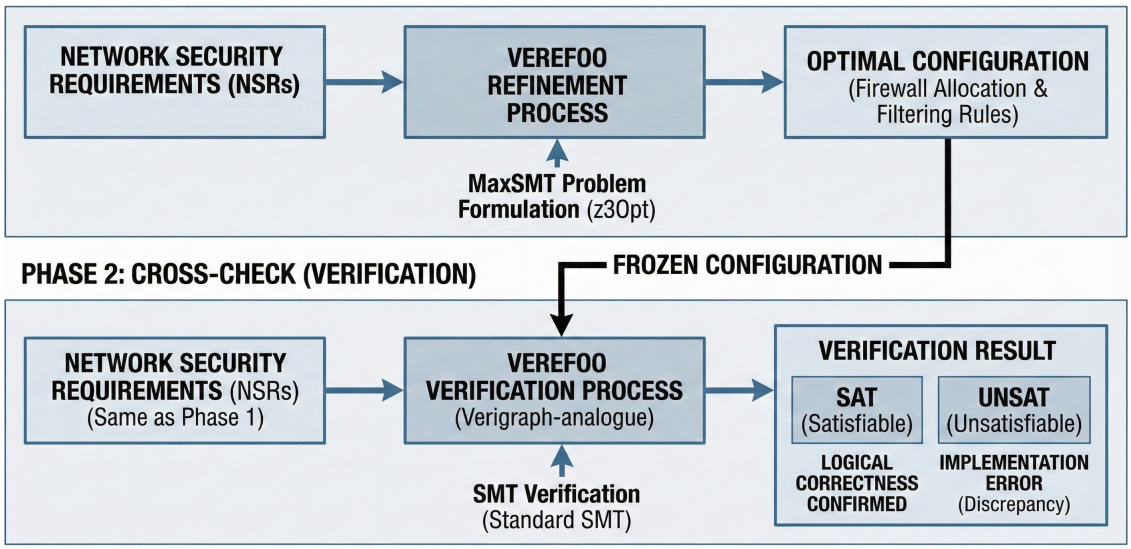


Figure 4.1: Flowchart of the cross-validation methodology used for correctness validation. The process is divided into a *Refinement* phase, which generates the configuration, and a *Verification* phase, which checks its correctness against the initial requirements.

4.2 Test case selection criteria

The criteria used for choosing test cases for the validation of correctness (Objective 3.2) is based upon the necessity of checking the most logically complicated parts of the framework, developed for the management of the semantics of states (Section 2.5).

Based on these criteria the following coverage objectives have been defined; these have been created in order to "test" the most critical aspects of the implementation of the framework:

- **NAT and State Inversion:** Verifies the correctness of the inversion of the 5-tuple (t^{-1}) after the rewriting performed by a NAT. An inversion failure would invalidate the functioning of conditional permissions (`ALLOW_COND`) [4].
- **Multi-firewall and Distributed Consistency:** Demonstrates that the MaxSMT model produces globally consistent configurations in topologies with multiple instances of firewalls, validating the theoretical advantage (described in Section 2.6.1) of eliminating the need for synchronization of run-time.
- **Multiple Flows and Logical Segregation:** Verifies that the application of the `ACTION ALLOW_COND` to a flow (for example, from client A) does not inadvertently permit the transit of unrelated flows (for example, the return traffic of client B) through the same instance of the firewall [5].
- **L4 Variants and Protocol:** Validates the robustness of the logic of matching and inversion on complex predicates, including specific ports, port ranges and wildcard protocols (for example, `TCP`, `UDP`, `ANY`).

- **Expected Failure Test (Negative Test):** Verifies that the framework correctly returns `UNSAT` (unsatisfiability) when the input is deliberately unsatisfiable (for example, conflicting Reachability and Isolation requirements), demonstrating the robustness of the model.

Each test case was designed to represent a specific and recurring criticality in the real world use of stateful firewalls in distributed architectures, in order to exclude errors of implementation in the most complex formal logic of the framework.

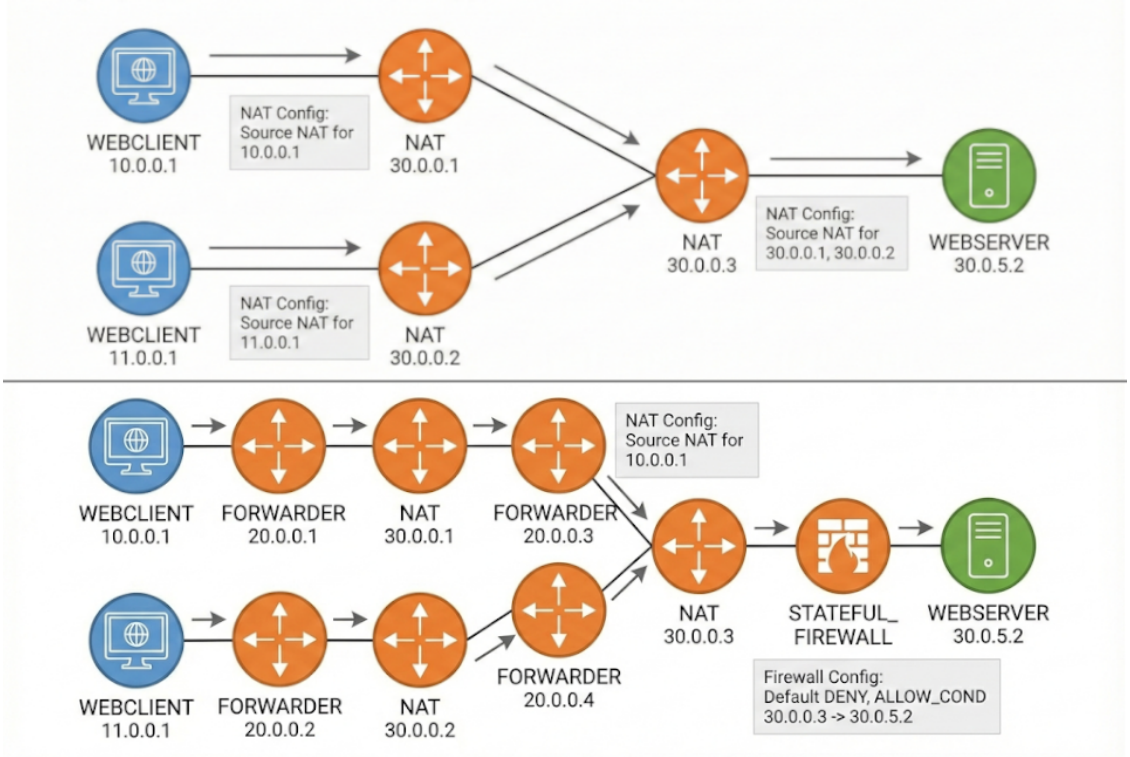
4.3 Inventory of tests

The suite of tests for the validation of the logical correctness of the framework was developed to systematically cover all the criticalities identified in Section 4.2. The tests are organized into categories of function, each aimed at validating a specific aspect of the VEREFOO formal model for stateful firewalls. All tests are located in the `StatefulCorrectness` directory and use XML files as input for the framework.

Tests Basic (Validation of a single NSR) Before dealing with complex scenarios, it was necessary to validate the behavior of the framework on minimal topologies with a single type of requirement at a time. `SimpleReachabilityProperty.xml` validates the generation of a stateless `ALLOW` rule for a simple reachability requirement between two nodes (`WEBCLIENT` → `WEBSERVER`), where traffic is permitted in general terms. Instead, `StrongReachabilityProperty.xml` is focused on the concept of initialization of the connection: it verifies that the traffic flows from the source to the destination specifically as a forward flow (`NEW` state), distinguishing itself from simple reachability as it does not authorize such traffic to be simply a response. `IsolationProperty.xml` validates that the framework correctly generates `DENY` rules or uses the default action to block unwanted traffic. `ConditionalReachabilityProperty.xml` is a test designed to be intentionally `UNSAT`: an isolated `ConditionalReachabilityProperty` requirement (without the corresponding forward flow initializing the connection) cannot be satisfied, and therefore this test verifies that the z3 solver correctly returns `UNSAT`. The `ConditionalReachabilityProperty2.xml` file extends the previous test by introducing the `ReachabilityProperty` necessary to make the conditional requirement `SAT`; this property, in fact, authorizes the initial traffic flow, allowing the creation of the related entry in the State Table and therefore making possible the conditioned response. `WS2WCIsolation.xml` tests the isolation in the opposite direction (Server → Client), verifying the symmetry of the model.

Tests NAT (Validation of 5-Tuple Inversion) This category is the most important because NAT introduces address and port rewriting, therefore it requires the correct 5-tuple inversion (t^{-1}) for return traffic [4]. `NAT.xml` is the most complex test: it uses a 6-node topology with 3 NATs (Figure 4.3) and verifies that the framework correctly calculates the composition of series NAT transformations, also managing the flows between nodes behind different NATs. `NAT2.xml` validates the management of multiple flows through the same NAT (two clients towards a server), verifying that stateful rules do not generate interference

and that the mixing of different semantics (stateful for one client, stateless for the other) is managed correctly. `NAT3.xml` extends `NAT2.xml` by introducing a second server, testing whether the NAT correctly distinguishes the flows toward different destinations and if the 5-tuple inversion takes the destination address into consideration.



Bottom: VEREFOO output.

Bottom: VEREFOO output.

Figure 4.2: *Top:* Service graph of the `NAT.xml` configuration.
Bottom: VEREFOO output.

Tests with Multiple Flows (Validation of Logical Segregation) These tests verify that the stateful rules for the concurrent flows toward the same firewall do not generate security interferences [5]. `3Node_S-S-Reachability_Isolation.xml` tests two stateless clients toward the same server, with return traffic isolation, verifying the absence of overly permissive rules. `3Node_S-STR-Reachability_Isolation.xml` is a critical test of interference between stateless and stateful semantics: one client has `ReachabilityProperty`, the other has `StrongReachabilityProperty`, but for both there is an `IsolationProperty` on return traffic. The test verifies that the framework allows the forward flows but rigorously blocks the responses, without generating overly permissive rules. `MultipleFlow_ConditionalStates.xml` is the main test for logical segregation: two client-server pairs traverse the same stateful firewall, and the test verifies that the rules `ALLOW_COND` use the full 5-tuple to distinguish the connections, preventing a client from

receiving traffic destined to the other. `4Node_ServerUpdateScenario_Conditional.xml` represents a realistic communication server-to-server scenario: clients communicate with a first server, which in turn communicates with a second server (for example, a database), testing the management of stateful communication chains.

Tests with Layer 4 Specification (Validation of Rule Matching) These tests validate the behavior of the formal model when the requirements specify protocols, ports or port ranges. `L4&PROTO_MultipleFlow_ConditionalStates.xml` is a variant of the multiple flow test with explicit Layer 4 specification: it verifies that the rules `ALLOW_COND` include exact protocol and ports (for example, `TCP src=12345 dst=80`), that the rules of isolation with port ranges do not conflict, and that the mixing of different protocols (`TCP`, `UDP`) is correctly managed. `L4&PROTO_ComplexScenario.xml` is the most complex test in the suite: 9 nodes (3 clients, 3 servers, 3 cascaded firewalls) with 18 heterogeneous requirements, each with L4 specification. It validates the consistency of multi-firewall configurations with protocols and port constraints, the management of edge cases (`ANY`, `TCP`, `UDP` protocols), and the robustness of the MaxSMT solver on problems with high combinatorial complexity [6].

Test with Complex Scenario (Validation of Logical Scalability)
`ComplexScenario.xml` shares the same topology as `L4&PROTO_ComplexScenario.xml` (9 nodes, 18 requirements, 3 firewalls), but abandons the transport layer details using wild cards for ports and protocols. The purpose of this test is to validate the ability of the framework to correctly manage the generic requirements (for example, "all the traffic from A to B is allowed") in a multi-firewall configuration, verifying that the absence of specifications on the transport layer does not generate ambiguity in the generation of the rules of state or conflicts with the policies of isolation in non-trivial scenarios.

In conclusion, the motivation and statistics of the overall suite Each test represents a criticality recurrent in the use of stateful firewalls in distributed architectures in the real world. The suite covers a wide range of complexities: from minimal topologies (2 nodes, 1 requirement) to realistic scenarios (9 nodes, 18 requirements, 3 firewalls). In total, the suite includes 16 distinct tests, with node numbers varying from 2 to 9, numbers of NSRs varying from 1 to 18, and number of allocated firewalls varying from 0 (unallocated graph, to be optimized) to 3 (already allocated graph, to be verified). The tests were designed to "test" the most complex logic of the framework (inversion of state after NAT, distributed consistency of multi-firewalls, segregation of flows concurrent, L4 matching) and to eliminate errors of implementation even on edge cases. In the following Sections (Section 4.4), some representative tests will be analyzed in detail, showing the XML input, the generated output and the analysis of the rules.

Table 4.1 reports the main characteristics of each test included in the suite.

Test Case	Category	Nodes	NSRs	FW	Validated Criticality
SimpleReachabilityProperty.xml	Base	2	1	TBD	Basic *stateless* rules
StrongReachabilityProperty.xml	Base	2	1	TBD	First *stateful* semantics (ALLOW_COND)
IsolationProperty.xml	Base	2	1	TBD	DENY rules and *default action*
ConditionalReachabilityProperty.xml	Base (UNSAT)	2	1	TBD	Expected failure test
ConditionalReachabilityProperty2.xml	Base	2	2	TBD	*Stateless/stateful* mixing (*edge case*)
WS2WCIsolation.xml	Base	2	1	TBD	Reverse isolation (model symmetry)
NAT.xml	NAT	6	3	TBD	Chain of 3 NATs, composite inversion
NAT2.xml	NAT	4	4	TBD	Multiple flows through 1 NAT
NAT3.xml	NAT	5	6	TBD	NAT + multiple destinations
3Node_S-S-Reachability_Isolation.xml	Multiple Flows	3	4	TBD	2 *stateless clients*, same *server*
3Node_S-STR-Reachability_Isolation.xml	Multiple Flows	3	4	TBD	*Stateless/stateful* interference
MultipleFlow_ConditionalStates.xml	Multiple Flows	5	4	1	5-tuple segregation, 2 *stateful* flows
4Node_ServerUpdateScenario_Conditional.xml	Multiple Flows	4	6	TBD	*Client-server-server* chain (*stateful*)
L4&PROTO_MultipleFlow_ConditionalStates.xml	Layer 4	5	6	1	L4 *Matching* (ports, protocols, *range*)
L4&PROTO_ComplexScenario.xml	Layer 4	9	18	3	Complex *multi-FW* scenario with L4 constraints
ComplexScenario.xml	Scalability	9	18	3	*Baseline* without L4 constraints (*wildcard*)

Table 4.1: Inventory and characteristics of the Logical Correctness test suite.

Legend:

- **Nodes:** Total number of nodes in the topology (client, server, NAT, firewall).
- **NSRs:** Number of Network Security Requirements specified in the input.
- **FW:** Number of firewalls already present in the input graph. "TBD *(To-Be-Defined)*" indicates that the graph is unallocated (`serviceGraph="true"`) and the framework must decide where to allocate firewalls during the refinement phase. A specific number (1, 3) indicates that the graph is already allocated (`serviceGraph="false"`) and the test validates only the configuration correctness.
- **Validated Criticality:** Main technical aspect tested by the case.

4.4 Representative tests

The purpose of this section is to provide detailed information about four representative test cases that were developed to prove the hardest challenges of the VEREFOO formal model for stateful firewalls. Each test contains the topology, the critical needs, and the validation logic associated with each test.

4.4.1 `NAT3.xml`: 5-Tuple Inversion and Multiple Destinations

This test case is used to verify the correctness of the 5-tuple inversion (t^{-1}) after a NAT rewrite operation, while also managing the combination of stateful and stateless semantics.

Operational Scenario The operational topology is composed of 5 nodes: 2 clients (10.0.0.1, 10.0.0.2) connected to a NAT (30.0.0.1) which performs Source Network Address Translation (SNAT); 2 destination servers (30.0.5.1, 30.0.5.2); and a single firewall. There are six heterogeneous Network Security Rules (NSRs):

- Client 1 $\rightleftharpoons$ Server 1: Stateful bidirectional semantics (Strong + Conditional).
- Client 2 $\rightarrow$ Server 1: Stateless unidirectional semantics, with explicit return isolation.
- Client 2 $\rightleftharpoons$ Server 2: Stateful bidirectional semantics.

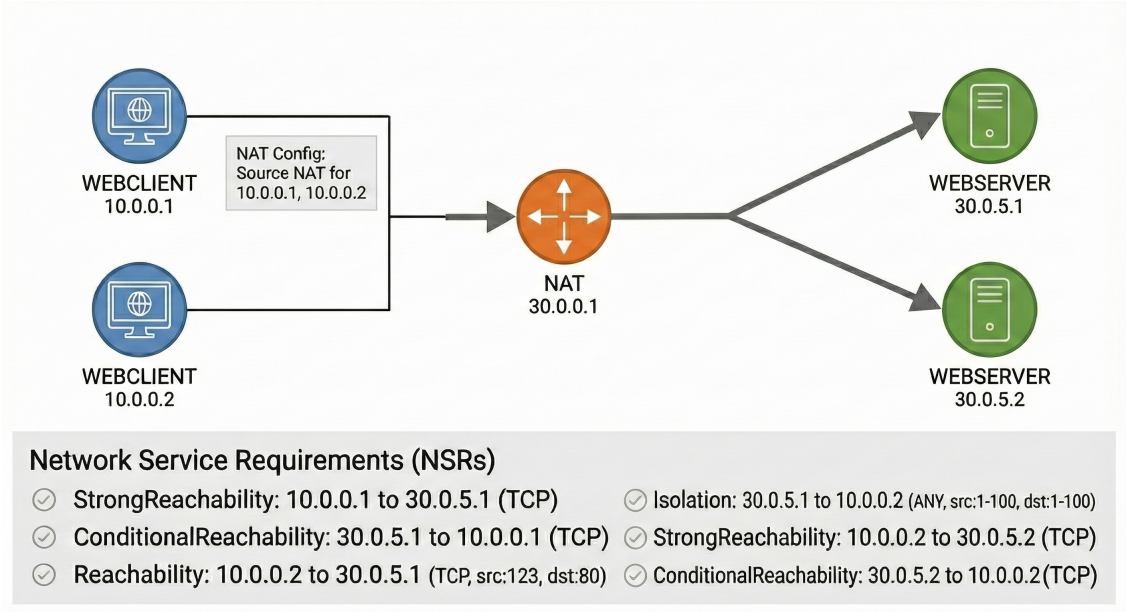


Figure 4.3: Topology of test `NAT3.xml`: post-NAT 5-tuple inversion towards multiple destinations.

Validation Logic The test case evaluates three major aspects of the formal model:

1. **5-tuple inversion:** The model must correctly compute t^{-1}, taking into account that the source address has been rewritten by the NAT (it becomes 30.0.0.1) [4].
2. **Destination disambiguation:** Because Client 2 communicates with two different servers via the same NAT, the framework must use the complete 5-tuple (including the original destination IP) to identify the return flow; otherwise, a routing ambiguity could occur.
3. **Hybrid Isolation:** The test verifies whether the `ALLOW_COND` rule generated for Server 2 is not over-permissive and does not incorrectly allow return traffic from Server 1 (which must be rejected).

Expected output: The expected output is SAT in both phases (Refinement and Verification).

4.4.2 `MultipleFlow_ConditionalStates.xml`: Simultaneous Flow Separation

The purpose of this test case is to prove that there is no logical interference among multiple concurrent stateful flows passing through the same firewall instance.

Operational Scenario Two client-server pairs (10.0.0.1 $\rightleftharpoons$ 30.0.5.4 and 10.0.0.7 $\rightleftharpoons$ 30.0.5.6) exchange data through a central firewall (20.0.0.3) with a previously allocated

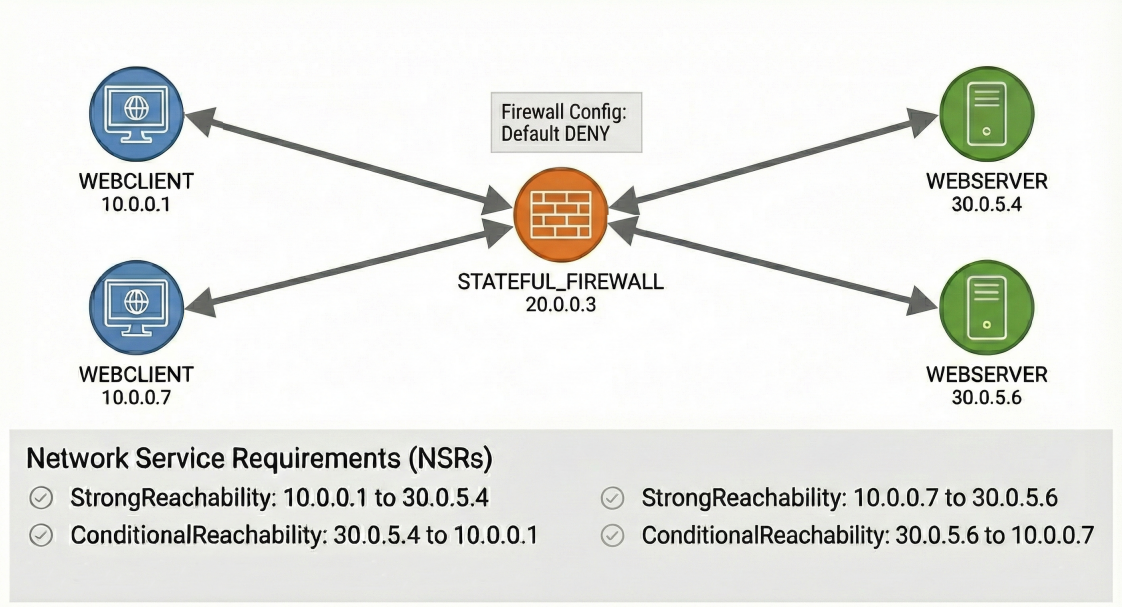


Figure 4.4: Topology of test `MultipleFlow_ConditionalStates.xml`: concurrent flow segregation.

5-node graph. Four symmetric stateful NSRs (Strong + Conditional) are enforced for each pair.

Validation Logic The test simulates a possible attack scenario where a malicious (or compromised) server tries to inject traffic into another session. Validation is focused on the specificity of the generated `ALLOW_COND` rules:

- If Client A establishes a connection to Server A, the firewall generates a dynamic rule.
- If Server B sends a packet to Client A, the model must reject it.

For the test to pass, the framework must create constraints using the full 5-tuple. A constraint-based solely on source or destination IP would be too lax, allowing cross-talk among sessions.

Expected output: Expected output is SAT in both phases, proving that segregation is correct.

4.4.3 `L4 & PROTO_ComplexScenario.xml`: Distributed Consistency with Layer 4 Constraints

This test case is the most complex in the suite, developed to validate distributed consistency across a firewall chain without run-time synchronization, and to include specific Layer 4 constraints (ports and protocols).

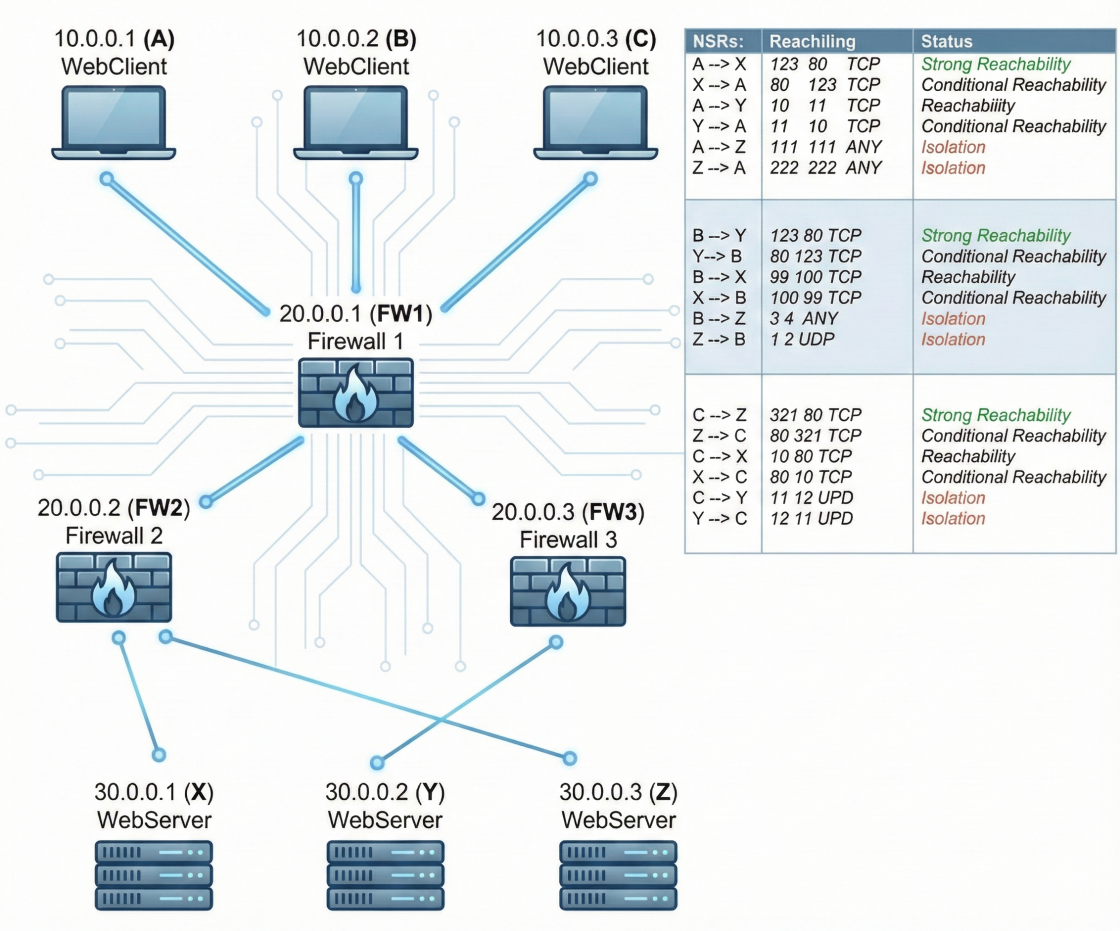


Figure 4.5: Topology of test `L4&PROTO_ComplexScenario.xml`: firewall chain with L4 constraints.

Operational Scenario The operational topology is a 9-node graph with 3 Clients and 3 Servers, separated by a 3-firewall chain (a front-end and two back-end). The system must meet 18 heterogeneous NSRs that specify protocols (`TCP`, `UDP`, `ANY`) and particular ports (e.g., `src=123`, `dst=80`).

Validation Logic This test case emphasizes the ability of the MaxSMT solver to manage large combinatorial complexities by verifying four conditions:

1. **Distributed Consistency:** All firewalls involved in a flow (for example, 20.0.0.1 and 20.0.0.2) must agree on the `ALLOW_COND` rules generated during the refinement process, as they represent a single atomic solution [7].

2. **L4 Matching:** The generated rules must respect the specified ports and correctly perform inversion (example: request `dst:80` → response `src:80`).

3. **Protocol Hierarchy:** An `ANY` rule (generic for isolation) cannot replace or invalidate a `TCP` rule (specific for reachability) and vice versa.

4. **TCP / UDP Mixing:** The framework must distinguish flows based on their transport protocol and avoid conflict between UDP isolation rules and TCP permissions on the same node.

Expected output: Expected output is SAT in both phases.

4.4.4 `ConditionalReachabilityProperty.xml`: Failure Test (UNSAT)

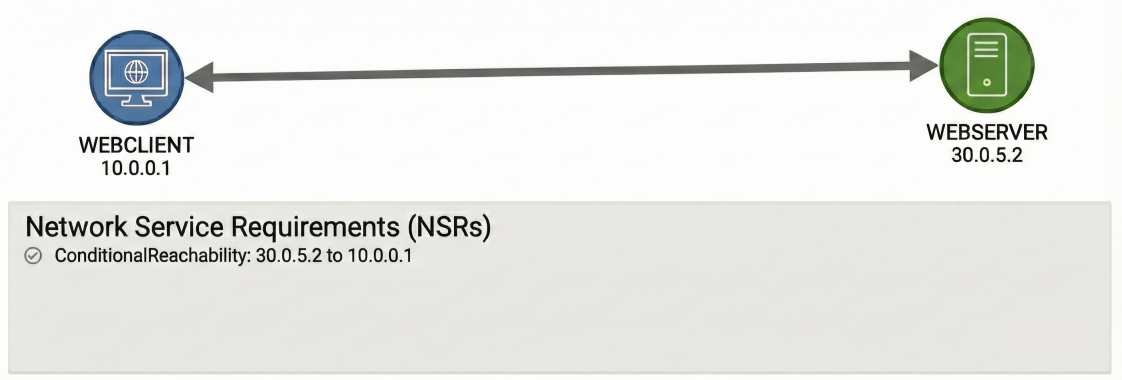


Figure 4.6: Representation of test `ConditionalReachabilityProperty.xml`: absence of the *forward* flow.

This failure test is essential to show the model's ability to detect logical inconsistencies and to verify that the framework does not produce "magic" solutions that violate the specifications when the required conditions are impossible to meet [6].

Operational Scenario The operational topology consists of only 2 nodes, one of them being a single requirement: a `ConditionalReachabilityProperty` from Server to Client. The `StrongReachability` property (the forward flow) necessary to initiate the communication is intentionally absent.

Validation Logic The test case creates a logical contradiction for the formal model:

- The requirement necessitates that return traffic be permitted (Server → Client).
- By definition, the `ALLOW_COND` action requires that there exist a previous state, created by a forward flow, to permit the return flow.
- Since the forward flow is not permitted by any requirement, the state cannot be created.

Therefore, the z3 solver must find this unresolved circular dependency and report that the problem is unsolvable (`UNSAT`).

Expected output: Expected output is **`UNSAT`** in the Refinement phase. A SAT result indicates a serious logical error (violation of conditional semantics).

4.5 Conclusion

The test suite has enabled the validation of Objective 1 (Logical Correctness) according to the success criteria defined in Section 3.2.

The validation was performed using two major criteria:

- **Positive Case (Reachability/Isolation Test):** A test is successful (`SUCCESS`) if and only if Phase 1 (Refinement) returns SAT (an optimal configuration is found) AND Phase 2 (Verification) returns SAT (such configuration meets the NSRs).
- **Negative Case (Expected Failure Test):** A test designed to be unsatisfiable (like `ConditionalReachabilityProperty.xml`) is successful (`SUCCESS`) if Phase 1 (Refinement) correctly reports UNSAT.

Successful completion of the entire test suite, that covers all the criticalities (Section 4.2) and representative cases (Section 4.3) of the formal model, proves the logical correctness of the formal model. These results demonstrate that the implementation of stateful semantics (Section 2.5) is correct and that the refinement process (Section 2.4) produces configurations that are consistent with the verification model.

With the logical correctness of the model demonstrated, the analysis may continue with the evaluation of Objective 2. The scalability and performance of the framework, and the effect of stateful constraints on complex scenarios, will be analyzed in Chapter 5 and 6 respectively.

Chapter 5

Scalability Test Case Design for Performance Evaluation

Chapter 4 has enabled the evaluation of the logical correctness of the stateful extension of the VEREFOO framework, validating through specific tests whether the configurations generated by the framework meet the security requirements without inducing inconsistencies. Formal correctness is just one of the two fundamental goals of this thesis; the other goal, which is no less important than the first, is the capacity of the framework to provide efficient operation on large-scale and complex networks [7].

This Chapter is completely devoted to the design and definition of the testbed (the test case) that will be used for the performance and scalability validation. In order to ensure that the experimental results obtained are meaningful and usable in real-world scenarios, generic synthetic topologies are insufficient; therefore, a model representative of the structural and operational difficulties of a modern critical infrastructure is needed [10, 16].

Section 5.1 illustrates the reference model chosen for this analysis: the Texas 2000. The characteristics of this cyber-physical system, representative of a Smart Grid electric network, will be shown, and the reasons why it is the ideal candidate to test the optimization potential of the framework, due to its intrinsic hierarchical complexity and criticality of its services, will be analyzed [2, 13].

Section 5.2 will analyze the network architecture from both the topological and functional points of view. From a simplified logical model, it will move to a complex and rigorously segmented topology, designed to respect industrial security standards (like NERC-CIP-005-7 [12]), in which the key components, demilitarized zones (DMZs) and security perimeters, will be recognized [9] that constitute the core of the test case.

Finally, Section 5.3 will focus on modeling the operational flows crossing such an infrastructure. Protocols typical of this area (industrial and management) (DNP3, ICCP, SQL, and HTTP) will be studied and it will be explained how these operational flows can be formally translated into a complete set of NSRs (Network Security Requirements), necessary to direct the policy generation process [1, 16].

5.1 The Texas 2000 Model: Context, Scope, and Relevance

5.1.1 Architectural Characteristics and Operational Context

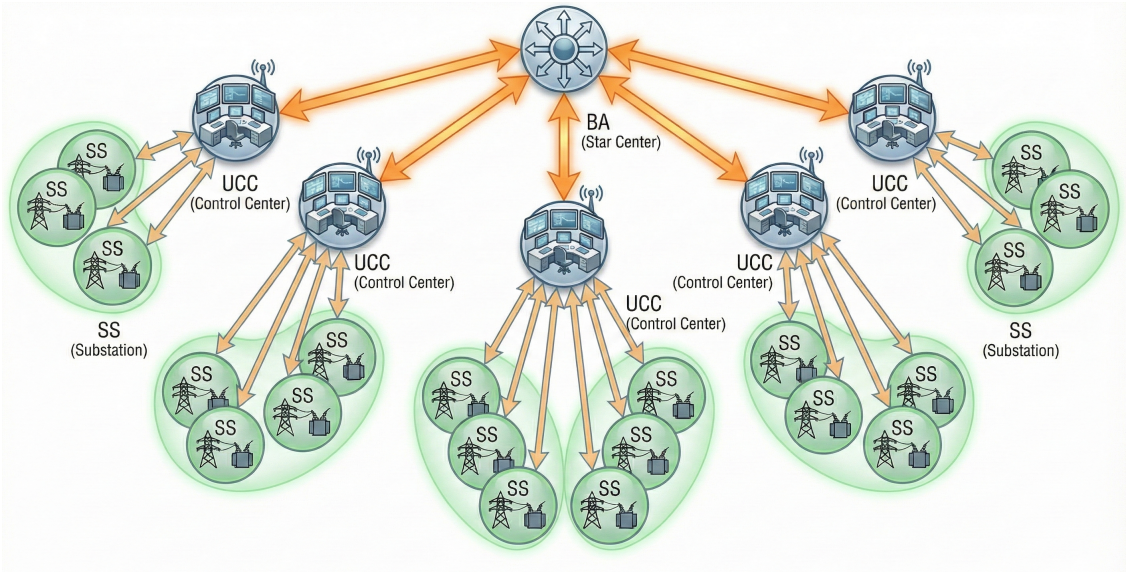


Figure 5.1: Schematic representation of the hierarchical topology of *Texas 2000*. The diagram illustrates the interconnection between Control Centers (UCC) and Substations (SS) *clusters*, highlighting the distributed nature of the model [13].

To validate the scalability (Objective 2) of the performance of the framework, it is essential to identify a test case that represents a realistic, complex, and large-scale environment. We have selected the Texas 2000 model. It is not a generic topology, but a synthetic yet plausible model of the Texas electrical transmission grid [2, 13].

Why did we choose this model? The main reason is its nature: it is an electrical network (a smart grid) – a mission-critical infrastructure [10]. Smart grids differ from corporate networks: they are complex CPS (cyber-physical systems) in which physical operations (electric energy distribution) are strictly controlled by a digital communication infrastructure [13].

The purpose of this model is to demonstrate the importance of security of this communication infrastructure for the stability of the physical grid. Therefore, reliability is regulated by strict standards (like *NERC-CIP-005-7*) [12], and this realistic model will allow us to verify VEREFOO's capability to generate stateful firewall configurations in a scenario requiring the rigorous network segmentation (e.g. isolation of Control Centers from Substations), required by such standards [10].

The Texas 2000 model, which is initially a physical model of buses and transmission lines, has been extended with a communication model, in which there are also cyber-components (routers, switches, firewalls, RTUs, IEDs) [13]. The resulting architecture is therefore inherently hierarchical and is based on two main types of logical nodes, which will serve as the basis of our scalability analysis (as described in Section 5.2):

- **UCC (Utility Control Center)** – regional control centers (or Balancing Authorities) from which grid operations are managed [13].
- **SS (Substations)** – electric substations located throughout the territory, containing physical devices [13].

Therefore, using this model enables testing VEREFOO's stateful refinement (which is necessary to represent industrial protocols such as DNP3, see Section 5.3) in a realistic, complex and large-scale topology, thus enabling a robust empirical validation of its computational scalability [7].

5.1.2 Suitability for the scalability benchmark

The choice of the Texas 2000 model (see Section 5.1) is not random and responds to a specific methodological necessity for the verification of Objective 2. As reported in Section 3.1, the major obstacle of VEREFOO is its theoretical computational complexity, which is NP-complete [3]. Moreover, this complexity is increased by the introduction of stateful semantics (see Section 2.5), which add a significant computational load for managing bidirectional flows and conditional constraints ($\otimes$) of Formula 2.4 [14].

A valid scalability test should therefore not be limited to increase the number of nodes, but must test the framework in a realistic, complex and large-scale scenario. The Texas 2000 model meets all three requirements:

- **Large Scale** – The Texas 2000 model, in its full representation, represents a large-scale electric transmission grid, possibly consisting of thousands of nodes (buses, substations and control centers) [2, 13]. Even though, for the purposes of our experiments, we will utilize a parametric generator (see Section 5.2), the ability to scale on this topology provides a clear evidence of the framework's computational feasibility (usability) in non-trivial cases.
- **Topological Complexity** – Unlike simple linear topologies (chains) or full mesh, the Texas 2000 model exhibits a complex hierarchical topology (Control Centers / UCC manage clusters of Substations / SS). Protecting this infrastructure requires a rigorous network segmentation (i.e., e.g., DMZ zones for isolating the control network from the corporate network or from external access) [9]. This topological complexity increases the number of possible paths and the difficulty of the allocation and configuration problem (MaxSMT solver) to be solved.
- **Requirements Complexity (Stateful):** Stateful semantics are the most relevant factor. A Smart Grid network uses only Web protocols, while it is based on industrial protocols (SCADA) like DNP3 or ICCP [13, 16]. These protocols are characterized by being bidirectional and therefore require the use of state tracking. For example, an ICCP communication between a server in a BA and a node in a UCC requires a forward flow for the request and a backward flow for the response. Such a scenario cannot be correctly represented with simple ALLOW actions (stateless), but requires the use of the stateful semantics introduced in Section 2.5, specifically

the combined use of `StrongReachabilityProperty` (for the forward flow I_i^{as}) and `ConditionalReachabilityProperty` (for the backward flow I_i^{ac}).

In conclusion, the Texas 2000 model is suitable for us as a scalability test because it allows measuring the simultaneous effects of a large-scale topology, a complex hierarchical structure and a set of realistic security requirements that heavily rely on the stateful extension of the framework. This will enable the empirical validation of the efficiency of VEREFOO's optimizations (see Section 2.6.2) in a mission-critical scenario compliant with industrial standards (*NERC-CIP-005-7*) [10, 12].

5.1.3 The Model's Logical Organization

The Texas 2000 model, selected as a case study for assessing scalability (Objective 2), relies on an electrical grid communication structure based on the same operational organization of actual power grids. The logical organizational hierarchy of the Texas 2000 model is centered around a star topology that can be divided into three logical levels (see Figure 5.1) [13]:

- **Balancing Authority (BA)/ISO:** The highest level is the central control unit (for example, ERCOT) that is in charge of the overall balance of the electric grid [13, 16].
- **Utility Control Center (UCC):** At the second level, regional control units supervise the different substations (via SCADA and EMS systems) [13].
- **Substations (SS):** At the third level, distributed physical plants contain the field equipment (for example, RTUs, relays) [13].

The central tenet of this structural design is separation: communication generally occurs vertically (from the BA down to the UCC and then to the SS) and rarely horizontally (between UCCs) [10, 16].

In terms of developing a cyber-physical model of this structure, there are two principal ways of representing the star topology found in the literature, each differing in level of detail and degree of compliance with industry regulation:

- **Simplified Functional Model (Model 1):** This representation, depicted in Figure 5.2, emphasizes functional connections [13]. It includes some form of segmentation (for example, a single DMZ for services that are openly available to the public (for example, a Web Server)) [9]. Although useful for the preliminary operational evaluation, this model does not represent the strict security requirements that are currently placed upon modern critical infrastructures.

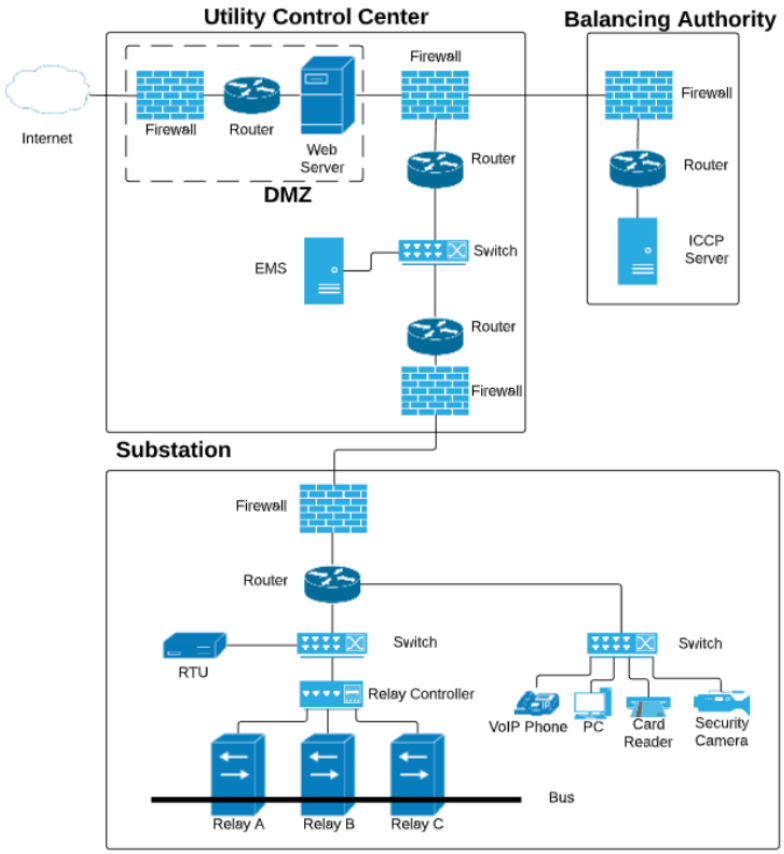


Figure 5.2: Functional Model [13], showing the basic topology of UCC, BA, Substation, and a single DMZ.

- ***NERC-CIP-005-7* Compliant Model (Model 2):** This model, shown in Figure 5.3, is an expanded version of the original model that reflects the need for regulatory compliance under the NERC-CIP-005-7 standard [12] that stipulates:

  > *"to provide management of electronic access to BES Cyber Systems by defining a controlled Electronic Security Perimeter for the protection of BES Cyber Systems from compromise" [12].*

  This model specifically defines the Electronic Security Perimeter (ESP) required for the segregation of the operational network (OT) and dictates that DMZs be implemented as the technical means of implementing an ESP, functioning as buffer zones to segregate Critical Cyber Assets from outside networks and from corporate networks (IT) and other OT zones [10, 12]. This model further expands upon the original model by adding additional specialized DMZs based on function (for example, SCADA DMZ, BA DMZ, Public DMZ, Substation DMZ).

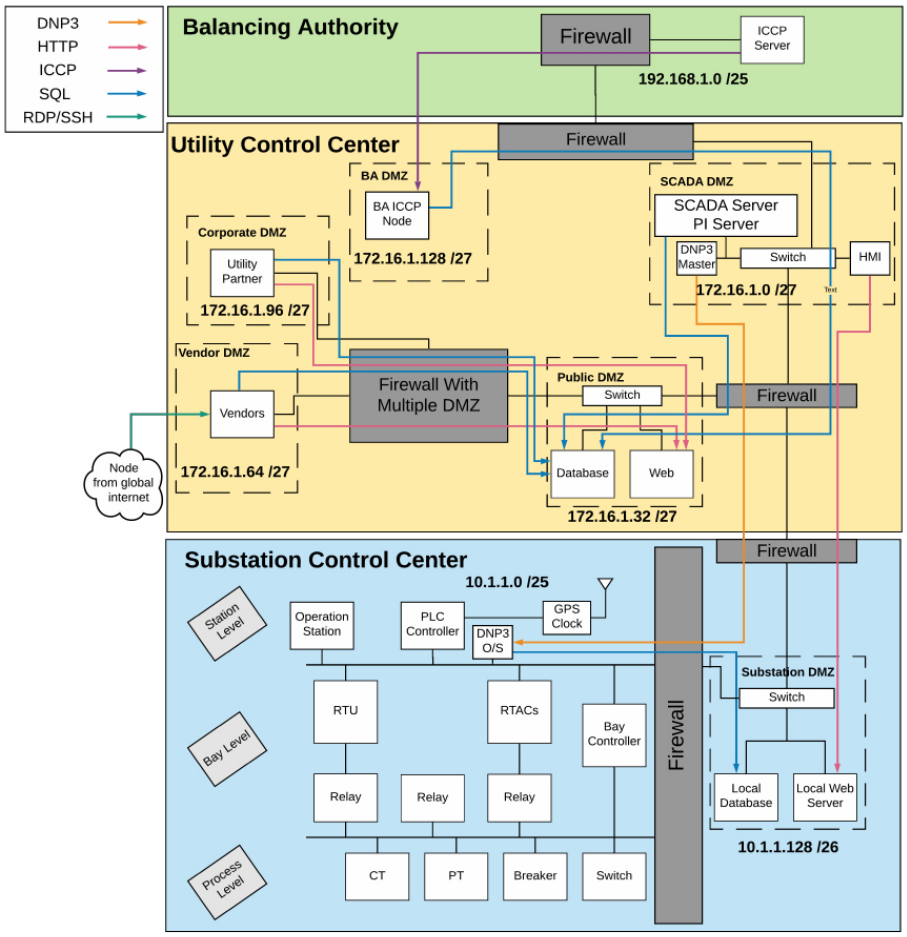


Figure 5.3: Complex *NERC-CIP-005-7* Compliant Model, showing the detailed topology with multiple firewalls and DMZs [9, 11].

A hybrid methodology was selected to pursue the goals of this dissertation. The foundation of the methodology was the Simplified Functional Model (Model 1) (Figure 5.2), which served as the topological basis for constructing the test case. From this base, only those components, protocols, and specialized DMZs that were necessary to construct a realistic test bed were added to the model incrementally, as described in the Complex Model (Model 2) (Figure 5.3). This hybrid methodology fulfilled two purposes: on the one hand, it allowed for the verification of the stateful semantics of industrial protocols such as DNP3 and ICCP; on the other hand, it allowed for the limitation of the computational complexity of the model in order to enable scalability analysis (Objective 2). Thus, the unnecessary overhead resulting from modeling the full compliant topology was eliminated.

5.2 Scalability Test Case Analysis

To develop a scalable test case (Objective 2) that is both realistic and relevant, a model (the simplified functional model) (Figure 5.2) of primary logical interconnections is insufficient [13]. That model is completely abstracted from the stringent network segmentation that is required for critical infrastructures by industry regulations [10]. Therefore, it is essential in this section to analyze the functional components of the basic model (Section 5.2.1), and then map them to a segmented and standards compliant network architecture (Section 5.2.2). This process is essential in order to demonstrate that the development of the basic topology toward the hybrid and compliant model that will be utilized for testing in Chapter 6, is justified. Our objective is to confirm that our test case is realistic enough to allow for the examination of industrial protocols (such as DNP3, ICCP) that will be examined in Section 5.3.

5.2.1 Functional Components of the Basic Topology (Simplified Model)

The basic architecture (Model 1) (Figure 5.2) presents the macro-components that define the primary functionality [13]. The identification of these elements is the first step to understand the primary flow of data. The functional components that are identified are:

- **ICCP Server (in the BA):** This is the exclusive component of the Balancing Authority that enables the management of the Inter-Control Center Communication Protocol (ICCP). It is the logical termination of the exchange of real time balancing data and telemetry between the utility control center (UCC) and the Balancing Authority [13].
- **DMZ (in the UCC):** The Demilitarized Zone represents an isolated area of the network (a buffer zone) within the Utility Control Center. As shown in Figure 5.2, the purpose of the DMZ is to house services that require exposure to external networks (for example, the Internet, or corporate networks), while protecting the internal control network [9, 16].
- **Web Server (in the DMZ):** It is the primary service housed in the DMZ of the Utility Control Center. It provides a data access interface (often via a web-based HMI) for remote operators, corporate partners (utility partner), or maintenance technicians (vendors) without direct access to internal critical systems [13].
- **EMS (Energy Management System):** It is the operational 'brain' of the Utility Control Center. The EMS is the software system that aggregates data received from the SCADA network, performs real time analysis, supports operator decision making, and transmits control instructions to the substation(s) [13, 16].
- **RTU (Remote Terminal Unit):** Located in the substation (SS), the RTU acts as the primary interface between the physical world (field devices) and the digital control network. The RTU collects measurement data from field devices (for example, relays) and transmits it to the Utility Control Center, typically using the DNP3 protocol [1, 13].

- **Relay Controller (in the substation):** It is the device that coordinates and manages intelligent protection relays (IEDs). It is responsible for the local automation of the substation (for example, performing rapid isolation of a fault detected by sensors (CT/PT) prior to receiving a command from the Utility Control Center [13]).

5.2.2 Development toward a Compliant Model (Complex Model)

Although the functional components identified in Section 5.2.1 are correctly organized, they are structured in a topology (Figure 5.2) that does not comply with current security standards for critical infrastructures. Reference Regulation *NERC-CIP-005-7* mandates the creation of secure boundaries (Electronic Security Perimeters) to protect "Critical Cyber Assets" (CCA) [12]. The segmentation created by the reference regulation results in the utilization of a multi-firewall architecture and the extensive utilization of multiple specialized DMZs as presented in the Complex Model (Figure 5.3) [9]. Therefore, to create a realistic test case that is capable of transmitting industrial protocol flows (that will be examined in Section 5.3), our hybrid topology must be developed to include this segmentation. We identify the following DMZs that derive from the Complex Model, that are applicable to the test case:

- **SCADA DMZ (in the UCC):** This is the most sensitive security zone. It creates a barrier to isolate the real time industrial control systems (SCADA Server, Historian (PI Server), DNP3 Master) from other areas of the network [9, 16].
- **BA DMZ (in the UCC):** A DMZ is created to isolate the communications that occur between the Utility Control Center and the Balancing Authority. It contains the BA ICCP Node, which acts as a gateway for the ICCP protocol, thereby preventing the Balancing Authority from having a direct connection with the internal SCADA network [9].
- **Public DMZ (in the UCC):** Replaces the single DMZ of the simple model. It houses the services that require (stringently controlled) access from external networks (for example, the Web Server and the public database (frequently a replica of the Historian)) [9].
- **Corporate/Vendor DMZ (in the UCC):** Additional network segments are created to establish individual access perimeters for corporate partners (Utility Partner) or maintenance personnel (Vendors). These nodes have limited access only to the Public DMZ [9].
- **Substation DMZ (in the SS):** The substation also has internal segmentation. This DMZ is established to segregate the local servers (for example, Local Database, Local Web Server) from the operational control devices (RTU, Relay Controller) so as to prevent local HMI access from compromising the protective network [9].

Implementing this segmented architecture (Model 2) is a necessity for our test case. This model establishes the security perimeters (firewalls) through which the industrial protocol flows (DNP3, ICCP, SQL, HTTP), that we will examine in the next section, must pass in order for us to specify realistic and compliant stateful NSRs.

5.3 Protocol Modelling & Requirements Definition (NSRs)

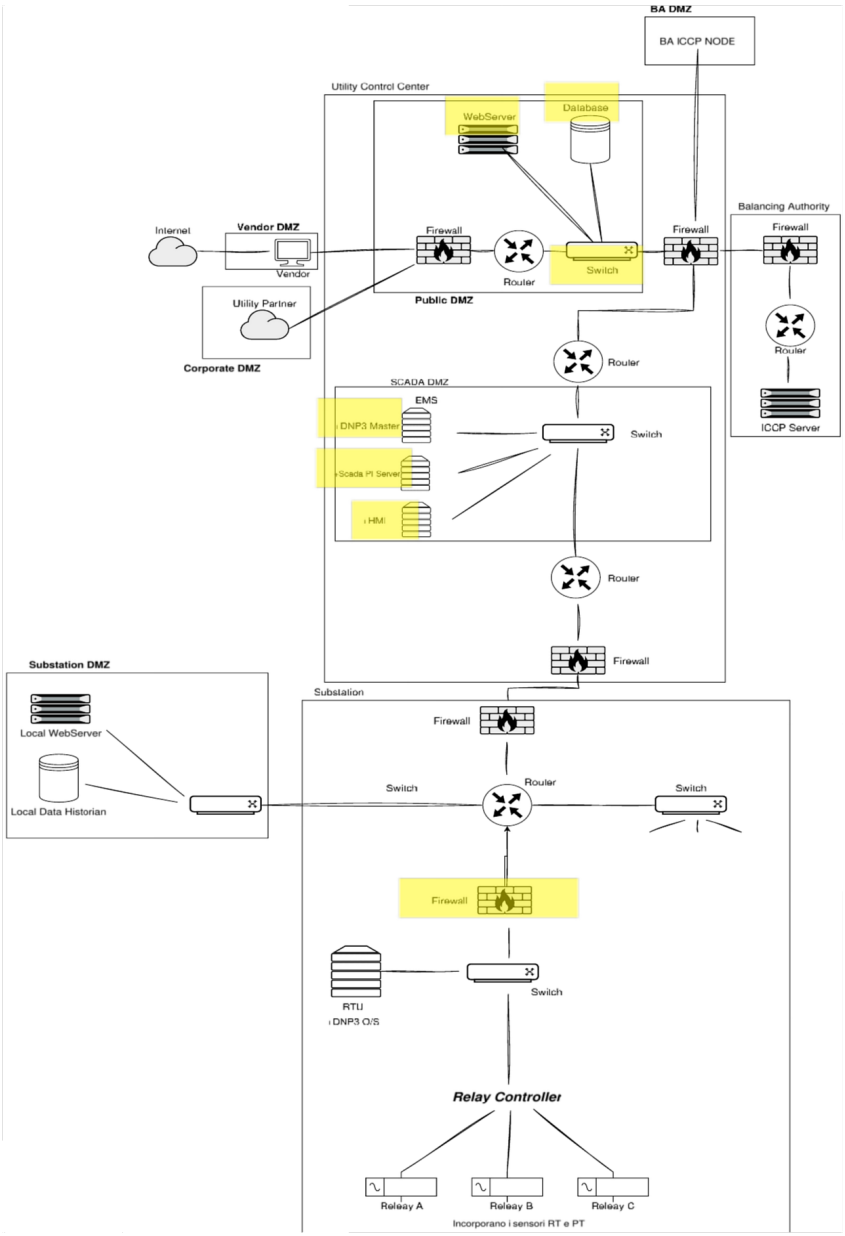


Figure 5.4: Hybrid Reference Topology used for protocol analysis and NSR definition. This architecture combines the hierarchical structure of Model 1 with DMZ segmentation (SCADA, BA, Public, Substation) and specific components of Model 2 [9, 13].

Following the description of the topological evolution of the system architecture from a simple functional architecture (Figure 5.2) to a segmented one (Section 5.2.2), the next step is to identify and describe how communications flow through this newly segmented

system. In this section we will:

- Identify the key network protocols typically found in an actual SCADA system [16].
- Develop Stateful Network Security Requirements (NSRs) that will be applied to validate against these networks protocols, as described above [16].

Here arises a methodological critical issue: industrial and management protocols (DNP3, ICCP, SQL) are used between specific components (e.g. DNP3 Master, PI Server, BA ICCP Node) that are easily identifiable in the Complex Model (*NERC-CIP-005-7* compliant) [9], but not necessarily always directly or homonymously represented in the simple model [13]. For example, the generic node "EMS" of Model 1 is logically related to the entire "SCADA DMZ" (which includes the DNP3 Master and PI Server) of Model 2 [9].

Thus, to create our test case, we must augment the simple model by creating equivalents to the missing nodes or adding segments (the DMZ) as derived from the Complex Model. This augmentation will allow us to achieve a hybrid topology that adequately represents the needed protocol flows.

Our Hybrid Reference Topology, which combines the Model 1 base model with the Model 2 segmentation and components, is illustrated in Figure 5.4. This figure will provide a common reference point for the analysis of all the protocols that follow (5.3.1 - 5.3.4), eliminating the need to depict the topology for each individual subsection [9, 13].

The subsequent subsections will individually review each key protocol in detail, referring to Figure 5.4 at all times to define the flow of the protocol and the derivation of NSRs:

- Section 5.3.1 - DNP3 Protocol (Distributed Network Protocol)
- Section 5.3.2 - HTTP/HTTPS Protocol
- Section 5.3.3 - SQL Protocol
- Section 5.3.4 - ICCP Protocol

5.3.1 DNP3 Protocol (Distributed Network Protocol)

The Distributed Network Protocol 3 (DNP3) is a foundational SCADA communication protocol utilized extensively in the energy sector for communications between Control Centers and Remote Components (distributed across the territory) [13]. DNP3 is built on top of TCP, utilizing standardized Port 20000 for IP communications. Its primary purpose is to facilitate Telemetry and Telecontrol, enabling Central Systems to monitor Asset Status and dispatch Operational Commands [1, 16].

In order to properly evaluate this protocol within our test case, it is essential to map the DNP3 Logical Roles (displayed in the Complex Model, Figure 5.3) to Functional Components already identified in our Simple Model (Figure 5.2) [9,13]. Since no additional nodes are added to the model for this mapping, functional roles are simply assigned:

- **Master (Initiator):** The **EMS (Energy Management System)**, already included in the Simple Model (Figure 5.2), plays the functional role of the DNP3

Master role. The reasoning behind this mapping is that the EMS is the operational "brain" of the UCC, responsible for Active Network Monitoring, Alarm Management and dispatching Real-Time Commands, which are precisely the types of operations of a DNP3 Master [1].

- **Outstation (Responder):** The **DNP3 Outstation (O/S)** role is played by the **RTU (Remote Terminal Unit)**, also present in the Simple Model (Figure 5.2). The RTU acts as a "Smart Bridge" in the Substation (SS), collecting raw data from Field Devices (e.g. CT/PT Sensors and Relays) and standardizing it for SCADA Communication towards the Control Center. Thus, it fulfills the exact role of a DNP3 O/S [1, 13].

Thus, the transition from the simple topology to our Hybrid Reference Topology (Figure 5.4) is achieved by relocating these existing components (EMS and RTU) into a *NERC-CIP-005-7* compliant network segmentation. As shown in Figure 5.4, the EMS (performing the DNP3 Master role) has been relocated to the **SCADA DMZ**, while the RTU (performing the DNP3 O/S role) is now located in the Substation Network and is separated from the rest of the network by a firewall to protect the critical operational traffic [10, 12].

The examination of the DNP3 communication flow reveals a Query-Response (Poll-Response) model, which is inherently Stateful [1]. A real-world example illustrates this behavior:

1. A physical fault (short-circuit) occurs on a substation line, which is detected by sensors and relays.
2. The RTU (DNP3 O/S) registers the event [1].
3. The **EMS (DNP3 Master)**, located in the UCC, regularly polls the DNP3 O/S (RTU) by transmitting a query request on Port 20000. This flow (Master → O/S) establishes the initial connection (**NEW** state) [1].
4. The RTU sends back the response (return flow O/S → Master) on the pre-existing TCP connection established earlier, transmitting the alarm data [1]. This response traffic is part of the **ESTABLISHED** state.

A stateful firewall, such as the one protecting the perimeter of the UCC (see Figure 5.4), should be able to manage the bi-directional and asymmetric nature of this flow. It should be configured to only accept connection establishment requests (`NEW`) from the UCC (trusted zone) to the SS. Once a connection is established, it should be configured to only permit response traffic (`ESTABLISHED/RELATED`) from the SS to the UCC, while rejecting any un-solicited `NEW` connection attempts from the substation.

The logic governing this stateful flow is expressed as a formalized set of Network Security Requirements (NSRs). Specifically, a `Strong Reachability` is established for the flow establishing the connection (Master → O/S), and a `Conditional Reachability` is established for the flow responding to the connection (O/S → Master), as detailed in Table 5.1 at the end of this chapter.

5.3.2 HTTP/HTTPS Protocol

HTTP (HyperText Transfer Protocol) and HTTPS (HTTP Secure) are industry-standard Application-Level Protocols for transferring data and facilitating User Interface interactions over the Internet. They use TCP, operating on Standard Ports 80 (HTTP) and 443 (HTTPS) [9].

As part of the Texas 2000 SCADA network, these protocols are not used for general public access; rather, they are vital for delivering Web-Based HMI (Human Machine Interface) interfaces to permit access to Diagnostic Dashboards or to permit Partners and Suppliers (Vendors) to view specific Data in a controlled environment [10].

To effectively represent these flows, we begin with the simple model (Figure 5.2), which identifies a generic "Web Server" in a "DMZ." However, the complex model (Figure 5.3) provides a stricter segmentation according to *NERC-CIP-005-7* standards, which we utilize in our hybrid reference topology (Figure 5.4). The topological evolution for this protocol is as follows:

1. **Public DMZ Segmentation:** Rather than including the Public DMZ in our hybrid model (Figure 5.4) as a singular entity, we include it as a segmented entity. The Public DMZ contains the **Web Server** and the **Database** as two separate nodes, connected to the same Internal Switch. Segmentation of the Public DMZ enables granular access controls [9].

2. **HMI Placement:** The `HMI (Human-Machine Interface)` is the primary terminal of the control center operator. Rather than a generic client, in our hybrid model (Figure 5.4) it is located, for security reasons, inside the **SCADA DMZ**, connected to the same Switch as the EMS (DNP3 Master) and PI Server [9].

Thus, the clients (vendors, partners, HMI) and servers (web server and local web server) for the HTTP/HTTPS flows are identified in Figure 5.4:

- **Clients:** `Vendors` (from Vendor DMZ), `Utility Partner` (from Corporate DMZ), and `HMI` (from SCADA DMZ).
- **Servers:** `Web Server` (Public DMZ) and `Local Web Server` (Substation DMZ) [9].

An examination of the flow of this protocol (real scenario) demonstrates the various uses of this protocol. While the mechanisms governing the statefulness of the protocol are similar, the actors and their interests (commercial and operational) are quite different:

- **Scenario 1 (External Access to Public DMZ):** This scenario describes controlled access by parties outside of the utility.
  - **Vendors (Suppliers)** are technicians from third party companies (for example, manufactures of relays or transformers). Their commercial interest is often under a maintenance agreement to perform remote diagnostics, examine logs or confirm the status of installed equipment. Thus, vendors access the `Web Server` in the Public DMZ.

  - **Utility Partners (Corporate Partners)** are neither equipment suppliers nor other companies/entities in the energy sector with which the UCC collaborates (for example, national transmission grid operators or market managers). Their interest is intercompany coordination, both commercial and operational, to balance electric generation and consumption. To coordinate this effort, utility partners require read-only access to aggregate energy information (for example, forecasted loads, current energy production). Utility partners access this information via the `Web Server` in the Public DMZ [9].

- **Scenario 2 (Internal HMI Access):** This scenario describes internal operational access for advanced diagnostics.
  - The **HMI** is the interface used by control center (UCC) operators who reside in the SCADA DMZ. Typically, the operator views the network via aggregate data received by the EMS via DNP3 (as discussed in 5.3.1).
  - **Fault Diagnosis Use Case:** Assume a recurring alarm ("glitch") is occurring on a relay in Substation X. The DNP3 data arriving at the central EMS may be polled (sampled) (for example, every 10-15 seconds) and lacks sufficient time resolution to diagnose the problem.
  - **Action:** The operator, via the HMI, initiates an HTTP/HTTPS session towards the `Local Web Server` residing in the Substation DMZ of Substation X.
  - **Advantages:** The Local Web Server serves as a secure interface to query the `Local Database` (Data Historian), which stores the events at high-resolution timestamps (for example, milliseconds). The operator can thus thoroughly analyze the details of the relay logs and accurately determine the root cause of the fault (for example, defective relay), which would be difficult to accomplish with only centralized aggregated data [16].

Regardless of the scenario, the Client (Vendor, Partner, or HMI) always initiates the connection (**NEW** flow) toward the Server. A stateful firewall that lies between the Client and the Server authorizes the request (slow path) and creates a state entry. When a Server sends back a response (**ESTABLISHED** flow), the stateful firewall automatically permits the response (fast path) because it correlates with a previously-created state entry, without requiring a static ingress rule.

The stateful flow logic is translated into formal NSRs that are required for validation. Both `Strong` and `Conditional Reachability` are defined for all participating actors (Vendor, Partner, HMI), referencing standard ports (80/443). The complete listing of NSRs for this protocol is presented in Table 5.1.

5.3.3 SQL (Structured Query Language)

SQL (Structured Query Language) is a query language and not a network protocol; however, in the context of this thesis, SQL represents the application-level protocol used by databases to transport queries. Due to widespread use in industry and SCADA environments for historical data management, it is assumed Microsoft SQL Server is

being utilized and is running on standard port 1433 [9]. SQL provides the means for operational and corporate actors to access structured operational, historical, or diagnostic data collected by the electric grid [10].

To properly map the flows, we utilize our Hybrid Reference Topology (Figure 5.4), which, based upon the segmentation described in the *NERC-CIP-005-7* model of the Complex Model (Figure 5.3), identifies two separate database nodes:

- **Database (Public DMZ):** This represents a centralized database server, located in the Public DMZ (isolated from the Web Server by a switch [9]), and acts as a central repository for historical and aggregated data. This data can be accessed in a controlled manner from multiple security zones.
- **Local Data Historian (Substation DMZ):** This represents a local database located within the Substation DMZ, and utilizes high resolution recording of substation events [9].

Flow Analysis (Real Scenario) indicates multiple use cases for the SQL protocol exist, and are all built upon a fundamentally stateful client-server interaction:

- **Scenario 1 (Local Data Recording – Substation):** The primary SQL flow within the substation is a write operation. The DNP3 O/S (RTU) is the client that sends SQL commands (i.e., "`INSERT INTO...`") to continuously record sensor data (voltage, current, switch status) in the Local Data Historian. This provides a high fidelity method of locally archiving data for immediate diagnostics.
- **Scenario 2 (Centralized Data Access – UCC):** The Database in the Public DMZ is the server that accepts requests from multiple clients with different operational and economic interests:
  - **PI Server (SCADA DMZ):** The PI Server acts as an SQL client to read historical data from the Public Database for aggregated analysis and visualization on the HMI (i.e., "Show me yesterday's load curve...") [9].
  - **Vendors (Vendor DMZ):** The vendors act as SQL clients to perform remote diagnostics, and query the Database for specific fault logs (i.e., "What relay generated the alarm at 08:32?") [10].
  - **Utility Partners (Corporate DMZ):** The utility partners act as SQL clients to consult aggregated and anonymized data necessary for inter-company coordination and market analysis (i.e., "Download monthly consumption data...").
  - **BA ICCP Node (BA DMZ):** The BA ICCP Node acts as an SQL client to read updated operational data (i.e., current production) from the Database, and format this information and send via ICCP to the Balancing Authority.

All these scenarios involve the Client (RTU, PI Server, Vendor, etc.) establishing the connection (`NEW` flow) towards the Database Server (Public DB or Local DB) on port 1433. A stateful interposed firewall allows the Client's request (slow path) and creates a state entry. Any subsequent response traffic (`ESTABLISHED` flow), i.e., results of a `SELECT`

or `ACK` of an `INSERT`, are subsequently automatically permitted (fast path) since they correlate with the previously created state.

This stateful flow logic is translated into a series of formal NSRs to govern database access. Each of the identified interactions (Vendor, Partner, PI Server, BA ICCP Node, and RTU) define the pair of rules to initialize and respond on port 1433. Details regarding these requirements can be found in Table 5.1.

5.3.4 ICCP Protocol (Inter-Control Center Communications Protocol)

ICCP protocol (Inter-Control Center Communications Protocol), also referred to as TASE.2, represents the industrial standard for exchanging real time data among different energy control centers. This protocol runs on the ISO/OSI stack, typically encapsulated in TCP. Within the context of the Texas 2000 system, the ICCP protocol is critical to vertical communication between the Utility Control Center (UCC) and the Balancing Authority (BA) [13, 16]. The ICCP protocol facilitates communication between the BA and UCC to enable the BA to observe UCC operational data, and transmit setpoints (load or generation targets) to maintain balance and stability of the entire inter-regional electrical grid [10].

To map this flow onto our Hybrid Reference Topology (Figure 5.4), we have identified the two logical endpoints developed from the Complex Model (Figure 5.3):

- **Endpoint A (Server):** The `ICCP Server`, residing in the Balancing Authority (BA) network [13].
- **Endpoint B (Client):** The `BA ICCP Node`. This component was introduced during the evolution from Model 1 and has been placed in a unique DMZ, the **`BA DMZ`**, which resides within the UCC perimeter [9]. This placement is a significant *NERC-CIP-005-7* requirement to provide an additional layer of security and segregate traffic entering from outside the control center (the BA) from the internal control network (SCADA DMZ) [12].

Flow analysis (real scenario) demonstrates how the ICCP protocol is used for operational coordination. The parties involved include the Balancing Authority (BA), acting as regional grid manager (e.g., ERCOT), and the Utility Control Center (UCC), acting as local operator. The **Balancing Setpoint Sending** use case illustrates the process: The BA operator establishes that the area managed by the UCC needs to adjust its load profile (e.g., “new setpoint of 90MW requested”). The process is as follows:

- **Communication Flow:** The command is transmitted from the `ICCP Server` (BA) to the `BA ICCP Node` (UCC) utilizing the ICCP protocol (TCP).
- **Isolation and Integration:** To adhere to the *NERC-CIP-005-7* isolation requirement, the `BA ICCP Node` (residing in the `BA DMZ`) does not communicate directly with the `EMS`. Instead, it acts as a secure proxy, receiving the ICCP command, translating it, and inserting the new set point (90MW) into the `Database` (located in the `Public DMZ`) utilizing the SQL protocol (as illustrated in Section 5.3.3).

- **Action:** The `EMS` (from the `SCADA DMZ`) reads this updated value from the `Database` and makes this value available to the `EMS` for operational decisions. Ultimately, the `EMS` translates the set point into specific DNP3 commands to be sent to the substation(s).

The interaction is a typical TCP session. The `ICCP Server` (BA) initiates the connection (`NEW` flow) on port 102 towards the `BA ICCP Node` (UCC). The firewall providing protection to the `BA DMZ` permits the `BA ICCP Node`'s request (slow path), creating a state. The `BA ICCP Node` uses the same connection (`ESTABLISHED` flow) to transmit acknowledgments or requested telemetry data (fast path).

This stateful flow logic is translated into corresponding formal NSRs. A `Strong Reachability` is established for the command from the BA to the UCC and a `Conditional Reachability` for return traffic on port 102. The formal specification for the above mentioned NSRs is provided in Table 5.1.

5.3.5 Summary Table of Network Security Requirements (NSRs)

As discussed in previous sections (5.3.1 – 5.3.4), the analysis of fundamental operational and management protocols (DNP3, HTTP/HTTPS, SQL, ICCP) present in our Hybrid Reference Topology (Figure 5.4), identified actors (clients and servers) in the respective protocols, analyzed real use cases, and evaluated the stateful nature of the communication flows.

This section combines these analyses by providing the complete set of Network Security Requirements (NSRs) that will serve as inputs to the Network Generator (covered in Chapter 6). As previously stated, generating a formally correct and optimal firewall configuration is contingent upon the VEREFOO framework's ability to meet these requirements.

Each NSR is defined as a tuple that includes the source (`Src`), destination (`Dst`), protocol (`Lv4`), destination or source port (`Port`), and requirement type (`Type`) to reflect stateful behavior:

- **Strong Reachability:** Utilized for the flow that initiates the connection (`NEW` flow). Represented by the firewall's `ALLOW` or `ALLOW_COND` action.

- **Conditional Reachability:** Utilized for the response flow (`ESTABLISHED` flow). Only satisfied when a state created by the forward flow exists.

- **Isolation:** Utilized to prohibit communication. Compliant with the rigorous *NERC-CIP-005-7* segmentation implemented through DMZs, the architecture utilizes an approach based upon the *least privilege* principle: traffic is permitted to travel only along functionally defined paths (*Reachability*). Therefore, to maintain the integrity of the security perimeter, any non-reachable node pair (and therefore not operationally required) shall be subject to an *Isolation* constraint. This prevents arbitrary connections, specifically those seeking to traverse zones of differing security levels without proper authorization [12].

Table 5.1 below represents the comprehensive list of NSRs derived from our analysis. This represents a specification document for the implementation of the Network Generator (covered in Chapter 6). During scalability testing, each time the generator introduces a new topological element (e.g., new UCC or SS cluster), we will reference this table to determine which corresponding NSRs need to be concurrently added to the Network Generator. By doing so, we will ensure that the complexity of the refinement problem grows realistically and proportionally, while ensuring consistent mapping between the topology and security requirements.

ID	Source (Src)	Destination (Dst)	Lv4	Port	Type (Action)
DNP3 Protocol (Section 5.3.1)					
DNP3_FWD	EMS (DNP3 Master)	RTU (DNP3 O/S)	TCP	dstPort: 20000	Strong Reachability
DNP3_RET	RTU (DNP3 O/S)	EMS (DNP3 Master)	TCP	srcPort: 20000	Conditional Reachability
HTTP/HTTPS Protocol (Section 5.3.2)					
HTTP_VND_FWD	Vendor	Public DMZ Web Server	TCP	dstPort: 443	Strong Reachability
HTTP_VND_RET	Public DMZ Web Server	Vendor	TCP	srcPort: 443	Conditional Reachability
HTTP_PRT_FWD	Utility Partner	Public DMZ Web Server	TCP	dstPort: 443	Strong Reachability
HTTP_PRT_RET	Public DMZ Web Server	Utility Partner	TCP	srcPort: 443	Conditional Reachability
HTTP_HMI_FWD	HMI	Local Web Server (SS)	TCP	dstPort: 443	Strong Reachability
HTTP_HMI_RET	Local Web Server (SS)	HMI	TCP	srcPort: 443	Conditional Reachability
SQL Protocol (Section 5.3.3)					
SQL_VND_FWD	Vendor	Public DMZ Database	TCP	dstPort: 1433	Strong Reachability
SQL_VND_RET	Public DMZ Database	Vendor	TCP	srcPort: 1433	Conditional Reachability
SQL_PRT_FWD	Utility Partner	Public DMZ Database	TCP	dstPort: 1433	Strong Reachability
SQL_PRT_RET	Public DMZ Database	Utility Partner	TCP	srcPort: 1433	Conditional Reachability
SQL_PI_FWD	SCADA PI Server	Public DMZ Database	TCP	dstPort: 1433	Strong Reachability
SQL_PI_RET	Public DMZ Database	SCADA PI Server	TCP	srcPort: 1433	Conditional Reachability
SQL_BA_FWD	BA ICCP Node	Public DMZ Database	TCP	dstPort: 1433	Strong Reachability
SQL_BA_RET	Public DMZ Database	BA ICCP Node	TCP	srcPort: 1433	Conditional Reachability
SQL_RTU_FWD	DNP3 O/S (RTU)	Local Data Historian (SS)	TCP	dstPort: 1433	Strong Reachability
SQL_RTU_RET	Local Data Historian (SS)	DNP3 O/S (RTU)	TCP	srcPort: 1433	Conditional Reachability
ICCP Protocol (Section 5.3.4)					
ICCP_FWD	ICCP Server (BA)	BA ICCP Node (UCC)	TCP	dstPort: 102	Strong Reachability
ICCP_RET	BA ICCP Node (UCC)	ICCP Server (BA)	TCP	srcPort: 102	Conditional Reachability

Table 5.1: Summary Table of Network Security Requirements (NSRs) for the Hybrid Topology.

Chapter 6

Test Case Implementation and Scalability Validation

Previous chapters have presented the technological environment (Chapter 1); the formal architecture of the VEREFOO framework (Chapter 2); and the overall objective of this thesis (Chapter 3). Chapters 4 and 5 addressed Objectives 1 and 2 respectively; Chapter 4 provided a demonstration of logical correctness (Objective 1) using a formal test suite that provides evidence that the stateful firewall configurations produced by the VEREFOO framework correctly enforce the designated Network Security Requirements (NSRs), therefore validating the absence of errors in the application of the complex logical formulas in the MaxSMT model (Chapter 4).

Chapter 5, has progressed further and introduced a new and fundamental element: starting from the *Texas 2000* model—a realistic but artificial representation of an electrical network within a smart grid —it examined individual functional components, mapped industrial protocols (DNP3, ICCP, HTTP/HTTPS, SQL) onto the segmented topology of a *NERC-CIP-005-7* compliant network; and finally defined the entire set of stateful NSRs (Table 5.1) regulating communication in a high-risk infrastructure (Chapter 5).

This chapter completes the experimental path of the thesis by providing a demonstration of the second objective of the thesis—Performance and Scalability Validation of the VEREFOO framework—where Chapter 4 demonstrated that the VEREFOO framework produces correct configurations and therefore this chapter is concerned with whether the framework can scale to handle increased problem complexity. This is due to the fact that the MaxSMT problem is inherently NP-complete and that the inclusion of stateful semantics (Section 2.5) imposes far greater logical constraints than the stateless model on the z3 solver.

Section 6.1 describes the Test Case Generator Architecture, which is the software tool that was developed to generate parametrically similar instances of the *Texas 2000* model automatically. Its design pattern, the encoding of the *NERC-CIP-005-7* compliant topology (with all DMZs identified in 5), the automatic generation of NSRs derived from Table 5.1, and the management of Isolation Properties through parameters like `withPorts`, `withIsolation`, and `isolationBound` will be discussed. In addition, Section 6.1 demonstrates how the generator transforms the abstract analysis of Chapter 5 into a concrete

and reproducible implementation.

Section 6.2 defines the Experimental Test Methodology used to validate scalability. The automated execution framework (`TestPerformanceScalabilityTexas2000`), the sampling strategy based on 100 runs for each configuration (UCC, SS) with deterministic seeds to ensure reproducibility, and the important management of potential issues related to the implementation (z3 garbage collection and need for a full reset between executions) will be explained. Furthermore, the three test modes that were designed to evaluate the effect of each of the different scalability metrics independently will be illustrated: increase in UCC nodes separately (keeping SS constant), increase in SS nodes separately (keeping UCC constant), and combined linearly increasing both UCC and SS nodes (keeping UCC:SS ratio constant).

Section 6.4 provides an evaluation of the Output Management, explaining the incremental JSON format that was used to collect data (average time, maximum time, minimum time, total number of SAT/errors, property count) and documenting the tests that were performed along with the associated flags.

Finally, Section 6.5 presents the results of the experimental tests, with a detailed analysis of the scalability graphs and interpretation of the observed performance, confirming (or highlighting the limitations of) the computational feasibility of the VEREFOO framework in a realistic and complex scenario.

6.1 Test Case Generator Architecture

6.1.1 Design Pattern and Generator Structure

To develop a test case for testing scale (Objective 2, Section 3.3), it is necessary to employ an alternative methodology of creating test cases to avoid the necessity of manually developing the XML files needed for these tests. As stated above, due to the Hybrid Reference Model (Figure 5.4), the segmentation of *NERC-CIP-005-7* and the hierarchical nature of the Texas 2000 model, statically defining test cases is not possible. Because the total number of nodes follows the equation $N_{nodes} = 3 + 17 \cdot N_{UCC} + 14 \cdot N_{SS}$, and because the number of NSRs increases in proportion to N_{SS} (Table 5.1), a test with 20 UCCs and 40 SSs would necessitate the management of over 900 nodes and many hundreds of security properties. Due to the explosion of combinatorics associated with the number of nodes and the number of security properties, the use of a programmatic test case generator has been deemed appropriate.

The software component developed to meet this need is the `TestCaseGeneratorTexas2000` class, implemented as a **Factory pattern**. The Factory pattern was chosen for its ability to generate families of complex objects (in this case, NFV instances conforming to the `nfvSchema.xsd` schema) that have a similar structure, but differ in terms of their dimensional parameters. The generator is also a "smart constructor," that, depending on the set of input parameters provided, generates a full topology and a consistent set of NSRs, thus removing the possibility of manual configuration errors, and providing for the scientific reproducibility of tests.

The generator's architectural structure consists of five primary components:

1. **Configurable Input Parameters:** The class constructor accepts six parameters that determine the complexity and semantic aspects of the test case being generated:

   - `seed` ($s \in Z$): Seed for the pseudo-random number generator (PRNG), used for deterministic assignment of IP addresses to each node. This parameter is essential for reproducing the same topology for a given set of input parameters.
   - `numberUCC` ($N_{UCC} \in N^+$): Number of Utility Control Centers to generate. Each UCC will replicate the structure of Figure 5.4, including all required DMZs (SCADA DMZ, Public DMZ, Vendor DMZ, Corporate DMZ, BA DMZ).
   - `numberSS` ($N_{SS} \in N^+$): Total number of Substations. SSs will be distributed uniformly among UCCs based on the relation $\text{SS\_per\_UCC}_i = \lfloor N_{SS}/N_{UCC} \rfloor$ for $i < N_{UCC}$, with the remainder assigned to the last UCC.
   - `withPorts` (`boolean`): If `true`, security requirements will include specific TCP/UDP ports (e.g., `dst_port` = 20000 for DNP3, `dst_port` = 443 for HTTPS). If `false`, wildcards ("*") will be employed to simplify the MaxSMT problem, but reduce the level of realism.
   - `withIsolation` (`boolean`): Determines whether the generator will generate Isolation Properties. If `true`, the generator will apply the algorithm presented in Section 6.1.3 to add isolation requirements between non-directly connected nodes.
   - `isolationBound` ($B_{iso} \in \{-1\} \cup N^+$): Specifies the upper bound of the maximum number of Isolation Properties. The value -1 means that there is no limit. The formal precondition of this parameter is:

     $$\neg\texttt{withIsolation} \implies B_{iso} = -1 \tag{6.1}$$

     Violations of this precondition will cause an `IllegalArgumentException` to be thrown in the constructor to prevent the creation of an invalid configuration that could produce erroneous results in experiments.

2. **Deterministic IP Address Generation:** The topology will require each node to have a unique IP address. The `createRandomIP()` method employs a rejection sampling technique (essentially a "try and retry" technique) that utilizes a hash set to determine if the randomly generated IP address is unique. The algorithm generates a random address using a PRNG (whose `seed` will be initialized to reproduce results) and then checks to see if the address is unique by checking the `allIPs` set. If a collision occurs, the algorithm discards the address and generates another one until a unique address is found.

   It is easy to demonstrate the effectiveness of this approach (average time complexity $O(1)$) to show that address generation will rarely require multiple attempts by calculating the collision probability with regard to the total number of IPv4 addresses ($M = 2^{32}$). To find the probability of at least one collision occurring when generating N addresses, we look at the complementary probability that there are no collisions ($P(\text{no\_coll})$).

We proceed in a step-by-step manner: When the algorithm needs to generate the i-th address, there are already $i-1$ addresses that have been assigned and therefore are "taken." For no collision to occur, the new address must be selected from the remaining addresses, which are $M-(i-1)$. The probability of success for this single insertion is expressed in Equation:

$$P(\text{success}_i) = \frac{M-(i-1)}{M} = 1 - \frac{i-1}{M} \tag{6.2}$$

The total probability of having no collisions during the entire generation of N nodes is the product of the success probabilities for each single step (from address 1 to address N), as illustrated in [8]:

$$P(\text{no\_coll}) \approx \prod_{i=1}^{N} \left(1 - \frac{i-1}{M}\right) \tag{6.3}$$

Using the approximation $1-x \approx e^{-x}$ (for small x), we may convert the product of Equation above to a sum of the exponents:

$$P(\text{no\_coll}) \approx \prod_{i=1}^{N} e^{-\frac{i-1}{M}} = e^{-\frac{1}{M}\sum_{i=1}^{N}(i-1)} \tag{6.4}$$

In the Equation above, the exponent includes the summation of integers from 0 to $N-1$. Using the Gauss Sum formula for the first k integers ($\frac{k(k+1)}{2}$), and setting $k = N-1$, we get the simplification in Equation:

$$\sum_{i=0}^{N-1} i = \frac{(N-1)N}{2} \approx \frac{N^2}{2} \tag{6.5}$$

By substituting the result of the Equation above into the previous one, we find $P(\text{no\_coll}) \approx e^{-\frac{N^2}{2M}}$. Then, by using the inverse approximation ($e^{-x} \approx 1-x$) once again to compute the collision probability $P(\text{coll}) = 1 - P(\text{no\_coll})$, we find the final expression:

$$P(\text{coll}) \approx \frac{N^2}{2 \cdot 2^{32}} \tag{6.6}$$

For the analyzed topologies ($N \leq 10^5$), the resulting value from the Equation above is very small (< 0.01). This provides mathematical evidence that collisions are rare events and that the algorithm will usually succeed on the first try, so the generation of addresses will not be computationally expensive.

3. **JAXB Mapping and XSD Schema Compliant:** The generator will utilize JAXB classes automatically produced from the `nfvSchema.xsd` schema (Section 2.3.1) to

programmatically construct the Java object tree representing the NFV. Each component of the hybrid topology (Figure 5.4) will be represented as an instance of the `Node` class with the related `FunctionalType` (e.g., `STATEFUL_FIREWALL`, `WEBSERVER`, `FORWARDER`). Adjacent nodes will be represented as `Neighbour` objects, and the NSRs specified in Table 5.1 will be added to the `PropertyDefinition` element as `Property` objects.

The use of JAXB ensures that the generated NFV will be formally compliant with the XML schema required by the VEREFOO framework, thereby eliminating the possibility of syntax errors that would prevent successful input parsing. The `generateNFV()` method will produce a complete `NFV` object, which can be converted to XML via the JAXB marshaller, or sent directly to the solver via `VerefooSerializer`.

4. **Hierarchical Topology Creation Logic:** The core of the generator lies in the `generateNFV()` method, which performs incremental topology construction based on a modular strategy:

   - **Balancing Authority (BA) Creation:** A single BA node is created, containing the ICCP Server, a router, and a firewall. This subgraph is identical to every test case and represents the hierarchical entry point of the model.
   - **Loop for Generating UCCs:** For each $i \in [0, N_{UCC})$, the generator will create 17 nodes for each UCC (as determined by the formula N_{nodes}), replicating the structure of Figure 5.4: ingress firewall, router, switch, SCADA DMZ components (EMS/DNP3 Master, PI Server, HMI), Public DMZ firewall and switch, Public DMZ web server and database, Vendor DMZ firewall and router, and finally the BA ICCP Node in the BA DMZ. Each node will be connected to its neighboring nodes via `Neighbour` objects, and the IP addresses will be stored in arrays (e.g., `dnp3_master[i]`, `pubdmz_web_server[i]`) for later use in generating NSRs.
   - **Loop for Generating SSs:** For each $j \in [0, N_{SS})$, the generator will create 14 nodes for each SS, consisting of the firewall chain (boundary firewall, router, internal firewall), control network switch, DNP3 O/S (RTU), Relay Controller and the three relays (A, B, C), endpoint network switch, local client, and finally the Substation DMZ switch and servers (local web server and Local Data Historian). The Substations will be assigned to respective Control Centers in a round robin fashion to ensure balanced topology. At the same time, key IP addresses (e.g., `dnp3_os[j]`) will be stored for use in protocol generation.

5. **Automated NSR Generation**

   Populating the `PropertyDefinition` element will be accomplished by an algorithmic process that transforms the specifications in Table 5.1 into Java objects. Three distinct loops will be executed by the generator to address different competence areas of the topology:

   - **SCADA Interconnection Loop (UCC $\leftrightarrow$ SS):** The first loop will iterate

over the set of Substations (SS) to establish DNP3 telecontrol flows. The algorithm will implement a round robin distribution logic to assign each Substation to a reference Utility Control Center (UCC). For each identified pair (UCC-Master, SS-Outstation), strong reachability (command flow) and conditional reachability (response flow) properties will be created, establishing specific ports (TCP/20000) or wildcards depending on the input parameters.

- **Management and External Access Loop (UCC-Centric):** The second loop will iterate exclusively over UCC nodes to define internal and external access policies. For each control center, massive amounts of rules will be defined for accessing public dmz services (web server and database) by external actors (vendor and utility partner), and for iccp communication toward the balancing authority. This loop will manage the complexity of the heterogeneity of the various protocols (http, sql, iccp), utilizing the correct ports (443, 1433, 102) for each flow.

- **Local Recording Loop (SS-Centric):** The third loop will iterate over the set of Substations to configure flows contained within the substation dmz. In this loop, local sql traffic between the rtu (dnp3 outstation) and the local data historian will be permitted, allowing the persistence of operational logs directly in the field.

As a result, the total number of generated NSRs will grow linearly with respect to the dimensions of the generated topology:

$$N_{NSRs} = 2 \times N_{SS} + 12 \times N_{UCC} \tag{6.7}$$

(excluding Isolation Properties, treated in Section 6.1.3). The linear growth of the number of NSRs is intentional and corresponds to the hierarchical nature of the actual network: each additional topological component (UCC or SS) adds a constant number of communications to govern.

Table 6.1 presents the semantics and constraints of the generator parameters. This modular structure allows the generator to be both flexible (the parameters allow exploration of different regions of the scalability space), reproducible (the deterministic seed ensures that each test can be repeated), and compliant with the VEREFOO schema (utilization of JAXB eliminates syntactic errors). In subsequent sections, we will describe the algorithm for generating Isolation Properties (Section 6.1.3) and the implementation of the automatic test framework (Section 6.2), which utilizes this generator to execute hundreds of experimental runs.

Parameter	Type	Range	Semantics	Preconditions
`seed`	`int`	Z	Seed for PRNG (deterministic IP generation)	-
`numberUCC`	`int`	N^+	Number of Utility Control Centers	$N_{\text{UCC}} \geq 1$
`numberSS`	`int`	N^+	Total number of Substations	$N_{\text{SS}} \geq 1$
`withPorts`	`boolean`	{true, false}	If true, includes specific TCP/UDP ports in NSRs	-
`withIsolation`	`boolean`	{true, false}	If true, generates Isolation Properties	-
`isolationBound`	`int`	$\{-1\} \cup N^+$	Bound on number of Isolation Properties (−1=unlimited)	$\neg$withIsolation $\implies$ $B_{\text{iso}} = -1$

Table 6.1: Parameters of the Test Case Generator `TestCaseGeneratorTexas2000`.

6.1.2 Automatic Generation of NSRs

At the heart of security policy definitions resides the `createPolicy()` method, whose task is to generate formal constraints that comply with the VEREFOO XML schema from the theoretical flow analysis described in Table 5.1. In essence, `createPolicy()` translates the theoretical flow analysis into actual formal constraints. For each functional interaction identified in the topology, the generator creates, using JAXB factories, two `Property` objects: one, configured with the `name="StrongReachabilityProperty"`, for the initialization flow, and the other, configured with the `name="ConditionalReachabilityProperty"`, for return traffic.

With the goal of maintaining modularity and writing clean code, the generator has encapsulated the above mentioned logic inside the helper method `createPolicyPair()`. The `createPolicyPair()` method is responsible for managing the port configuration, based on the Boolean parameter `withPorts` (as anticipated in Section 6.1.1), which determines whether the generated constraints should be rigid or flexible. When the `withPorts` parameter is set to `True`, the method inserts specific numeric values of industrial protocols (for example, "20000" for DNP3) in the fields `srcPort` and `dstPort`, thus forcing the solver to meet a strict equality requirement. On the contrary, when the `withPorts` parameter is set to `False`, the wildcard character `*` is inserted; this action removes the transport port constraint, significantly reducing the z3 engine's search space.

6.1.3 Advanced Management: Isolation Properties

Besides the reachability requirements needed for the proper functioning of networks, another important aspect of security validation is the definition of isolation constraints (*Isolation Properties*). In fact, while reachability policies define what *must* happen, isolation policies define what *must not* happen, by preventing unauthorized lateral movements and segmenting the network into different security zones.

In the case of the Texas2000 use case, manually generating such constraints for thousands of nodes would be impracticable and prone to human errors. Therefore, a heuristic algorithm has been developed in the `createIsolationPolicies()` method,

with the purpose of automatically identifying candidate node pairs for isolation without contradicting previously existing functional requirements.

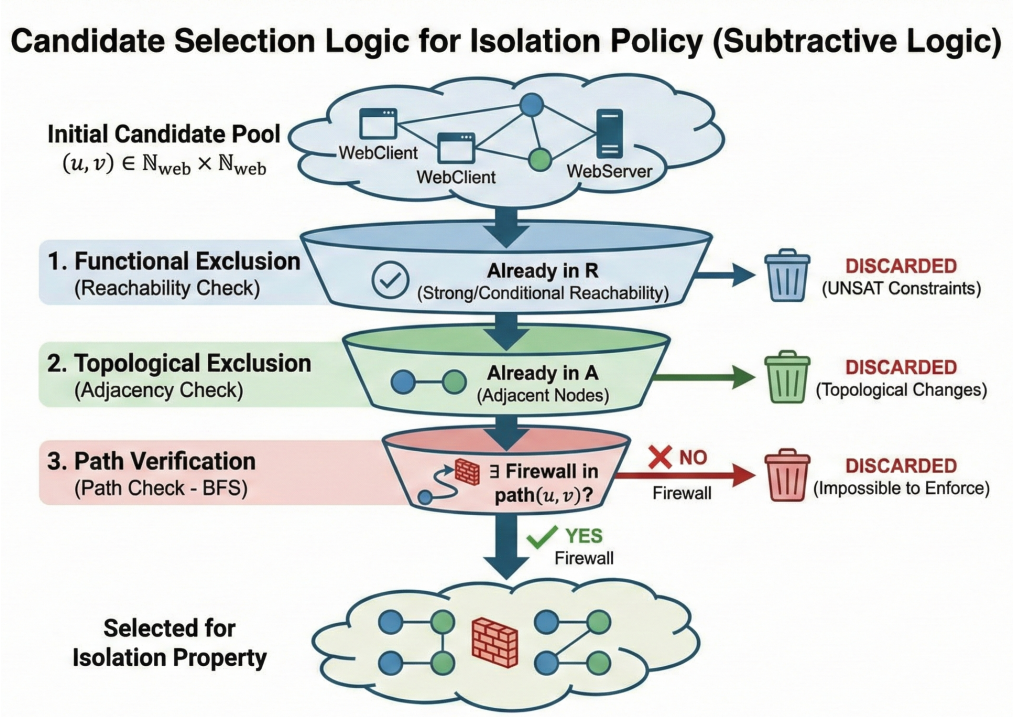


Figure 6.1: Illustration showing how the Isolation Policy is calculated by excluding all unnecessary cases.

The algorithm operates on a subset N_{web} of the graph nodes, i.e., the ones identified as `WebClient` or `WebServer`, since they represent the most critical targets and the most used attack vectors. The generation of isolation properties between two nodes $u, v \in N_{web}$ follows a subtractive logic, as follows.

Let R be the set of pairs (s, d) for which there exists a reachability property (Strong or Conditional) and let A be the set of adjacent node pairs in the topological graph (i.e., there exists a direct edge connecting them). A pair (u, v) is a candidate for an *Isolation Property* if and only if the following conditions are satisfied:

$$(u, v) \notin R \wedge (u, v) \notin A \wedge \exists \text{Firewall} \in \text{path}(u, v) \tag{6.8}$$

In practice, the generator performs the following filtering actions:

- **Functional Exclusion:** The pairs that need to communicate for SCADA operations (e.g. EMS to RTU) are eliminated, so as to avoid the creation of unsatisfiable (`UNSAT`) constraints a priori.
- **Topological Exclusion:** The adjacent nodes (the direct neighbors) are ignored, because the isolation between nodes that are physically connected and do not have

intermediary nodes would require topological changes that are not provided for in this test phase.

- **Path Check:** Using an optimized BFS (Breadth-First Search) visit, the algorithm checks if there is at least one node with Firewall functionality that exists in the path between the source and destination nodes. Without an intermediary enforcement point, the isolation request would be technically impossible to implement.

To control test scalability and to limit the number of constraints created during testing, the algorithm uses a limiting parameter called `isolationBound`. This parameter limits the number of generated isolation properties (K_{max}).

The algorithm iterates through the permutations of valid pairs and stops generating when the number of isolated properties $|P_{iso}|$ exceeds K_{max}. If the value of the parameter `isolationBound` is equal to -1, the generator works in *unbound* mode, creating constraints for all valid pairs found. This mechanism enables the modulation of the stress test on the z3 solver and enables the evaluation of how the degradation of performance occurs as the negative constraint density increases, as will be discussed in Chapter 7.

The entire logic of the algorithm is summarized in Algorithm 6.1.

Listing 6.1: Pseudo-code for Isolation Properties generation

```
Input: Graph G, Set<Property> ReachabilityProps, int K_max
Output: Set<Property> IsolationProps

Set<Node> WebNodes = filterWebNodes(G);
Set<Pair> Forbidden = extractPairs(ReachabilityProps) U getAllGraphEdges(G);
int count = 0;

foreach u in WebNodes do
    foreach v in WebNodes do
        if (u == v) continue;

        if (pair(u,v) in Forbidden) continue;

        if (!hasFirewallInPath(u, v)) continue;

        createIsolationProperty(u, v);
        count++;

        if (K_max != -1 AND count >= K_max) return;
    end
end
```

6.2 Experimental Test Methodology

The experimental assessment of VEREFOO scalability on the Texas 2000 topology entails a well-orchestrated set of experiments and evaluations. Simply running one experiment

and measuring the resolution time of the resulting instance does not suffice; the goal is to differentiate framework performance from intrinsic variability in the underlying z3 heuristic solvers.

Therefore, an automated test execution framework has been created and implemented in the `TestPerformanceScalabilityTexas2000` class to manage the lifecycle of the test, collect metrics and primarily to mitigate the effects of pseudo-randomly generated IP addresses (used to generate unique test cases) on the statistical reliability of the results of the experimentation.

6.2.1 Automated Test Execution Framework

`TestPerformanceScalabilityTexas2000` represents a parametric test harness and its operational logic iterates through a predefined set of test vectors (stored in the `testCases` array) defined in terms of dimensional configurations (N_{UCC}, N_{SS}).

To obtain statistically valid results, for each dimensionality configuration the system runs cycles of $N = 100$ independent runs. Each of the individual runs includes the following steps:

Instantiation: Invocation of the `TestCaseGeneratorTexas2000` described in Section 6.1, providing dimensional parameters and an initial seed.

Marshaling and Resolution: Conversion of the generated `NFV` object to XML format and invocation of the VEREFOO core (`VerefooSerializer`) to solve the MaxSMT problem.

Measurement: Calculation of wall clock time required to complete the resolution phase (excluding topology creation time), measured as the elapsed time from the moment the solver is invoked to the point when the `SAT`/`UNSAT` result is returned to the caller.

Incremental Logging: Immediately after resolving the MaxSMT problem, measurement values (wall clock time, number of nodes, number of properties) are written to a JSON file (`dataToPlot.json`). Writing measurement values incrementally avoids losing measurement data in case the system crashes during the course of a long duration test (which may last longer than 24 hours).

6.3 Implementation Challenges

6.3.1 The "seed" issue and Z3 heuristics

During the experimental testing period, it became apparent that the Z3 solver would sometimes need exponentially greater amounts of computation time to solve some problem instances that were topologically identical to other problem instances solved in a relatively short amount of time.

The reason for this behavior is related to the nature of SMT solver search algorithms and how sensitive they are to the internal representation of the problem. Although the

logical model abstracts away real IP addresses converting them into symbolic identifiers (atomic predicates), a different seed changes the order in which these data structures are created and assigned in the framework before being passed to the solver. As such, the solver receives the same logical assertions and variables, but in a reordered manner. Given that the solver uses heuristics to determine which branch to explore first based upon the order in which the variables are inserted into the search tree and the internal names of the variables, the solver will be placed in an inferior and possibly computationally expensive branch of the search tree based upon the input order.

When this occurs, simply generating a new instance using a new seed will modify the internal structure of the problem sufficiently so that the heuristic will select a faster and more efficient branch of the search tree.

6.3.2 "Randomized Restart" Strategy

To ensure that the experimental results represent true VEREFOO algorithm performance and not simply the "luck" of solving a particular instance of the problem, the test framework employs a **Randomized Restart** strategy.

As shown in the code control logic (the `while (!validRun)` loop), the test framework monitors the outcome of each execution. When an execution fails (returns `UNSAT` when it should have returned `SAT` or hangs) or the solver appears to have entered an infinite loop and timed out, the test is discarded rather than counted towards the final results. The test framework discards the current instance, generates a new random seed (`currentSeed = r.nextInt()`), generates a new topology using the new seed and launches another execution of the test.

This strategy insures that the data collected will be statistically significant and will accurately reflect the typical operating conditions of the VEREFOO algorithm, excluding pathological outliers due solely to low-level SMT solver heuristics.

It is worth noting however that the current implementation of this strategy is external to the VEREFOO core and was developed as part of the test framework to enable the realization of the experimental goals of this thesis. That said, given the importance of this strategy and the success achieved with the current implementation, the inclusion of a restart capability natively within the VEREFOO core is highly advisable in any future development of the VEREFOO engine, as outlined in Section 6.6.

6.3.3 Native Memory Management and Stability

As part of the extensive experimental validation, conducted under high-intensity iterative execution (100 runs per configuration) over a prolonged time frame (longer than 24 hours), a critical blocking condition arose from system resource management. During early iterations of the test session, the machine hosting the computation showed progressively increasing and severe performance degradation to the point of causing a complete operating system crash after 24 hours of uninterrupted operation.

An examination of error logs produced by the JVM revealed a seemingly contradictory anomaly: the Java process crashed due to failure of the JVM to allocate native memory at the operating system level (`native memory allocation (malloc) failed`); although

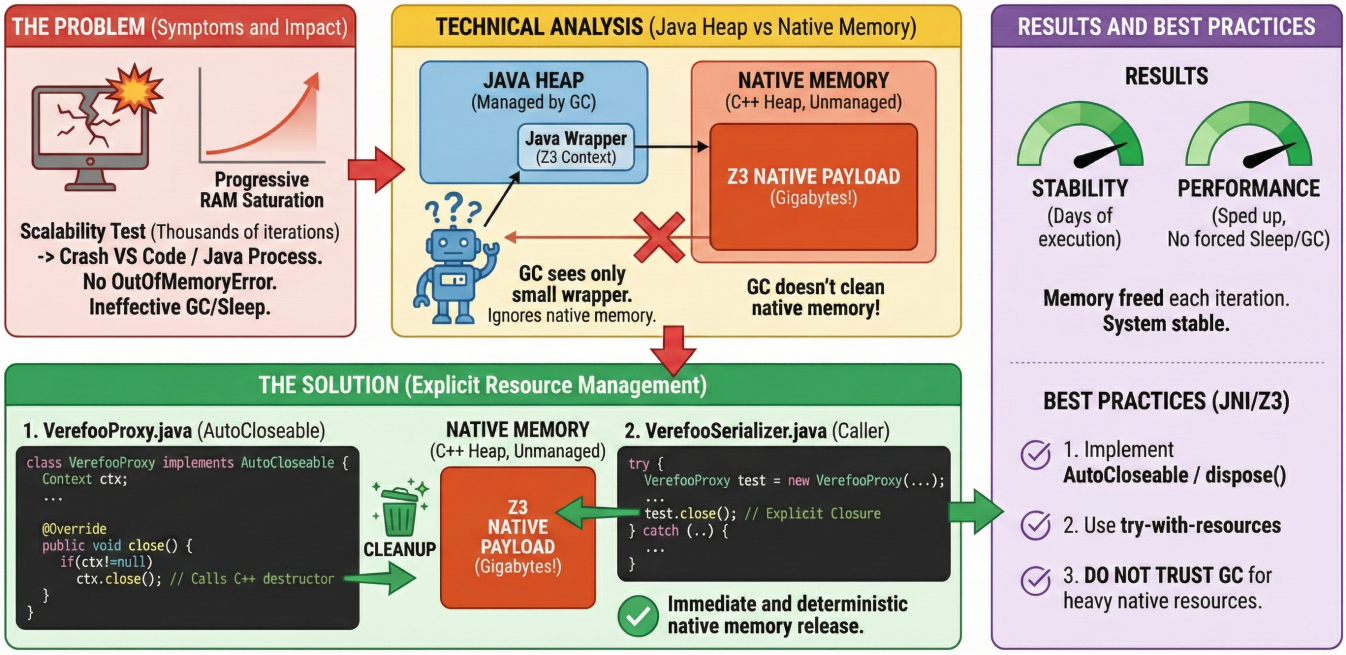


Figure 6.2: Z3 Memory Leak Fix in Verefoo: Cause and Solution

the Java heap (*Old Gen* usage $< 1\%$) was nearly empty and there was no indication of Java Heap exhaustion (`java.lang.OutOfMemoryError`).

Resource monitoring confirmed that both physical memory (RAM) and virtual memory (swap space) of the host system had reached maximum capacity. Technical analysis of the interaction between the JVM and the native Microsoft Z3 library employed by VEREFOO via JNI (*Java Native Interface*) found the source of the problem in the hybrid nature of Z3 objects:

- **Java Side:** The object representing the Z3 Context is very light weight (a few bytes).
- **Native Side (C++):** The object corresponding to the object representing the Z3 Context in the underlying computational engine creates very heavy-weight data structures in the *Native Heap* (up to several Gb for large instances).

Since the JVM's garbage collector (GC) only monitors the managed Heap, it did not perceive the impending system memory exhaustion caused by native objects. Thus, the garbage collector was not called frequently enough to free up the Java wrapper objects of the native objects, which resulted in orphaned native resources (Native Memory Leaks) that continued to accumulate over subsequent iterations until system crash.

To remove the structural impediment to the problem, the **Explicit Resource Management** pattern was applied. The `VerefooProxy` class was modified to implement the `AutoCloseable` interface and provide an explicit close method, which calls the C++ destructor of the Z3 context directly. The test framework was modified to ensure that

the `.close()` method is called deterministically for each computation iteration, releasing native memory immediately and avoiding reliance on the garbage collector.

This solution completely eliminated the problem of the saturation of the virtual memory pool of the host system. The subsequent scalability tests demonstrated complete stability of the framework and allowed the framework to execute continuously for days without showing any evidence of degradation in performance or unusual memory consumption, thus demonstrating the appropriateness of the solution for use in long-running enterprise environments.

6.4 Output Management

Output Management is the final step of the validation process which includes collecting all the experimental performance data systematically and structuring them. The output management is not just the output of the results on the screen, but also a structured way to persist the data to enable further analysis and visualization phases, especially given the high number of experiments needed to analyze the scalability of the framework. This section explains the selected format for serializing the experimental data and gives an overview of the different test setups validated.

6.4.1 Output Format

After each test run, the results are stored as a JSON formatted file, as defined by the JavaScript Object Notation standard. A single JSON object is created for each experiment and contains the metadata and the metrics of the experiment.

The metadata contain:

- **Configuration parameters**: Boolean values describing the context of the experiment (`withPorts` and `withIsolation`) and the bound imposed to the isolation properties (`isolationBound`).
- **Dimensional metrics**: The total number of nodes in the graph (`nNodes`) and the number of constraints generated (Reachability Property, Isolation Property and total).

The metrics contain:

- **Performance metrics**: The resolution times of the z3 solver averaged, maxima and minima values (over N runs) during the execution of the experiment.

In order to store the results incrementally, the output management is done per test configuration instead of storing all the results in memory until the end of the process. After completing the execution of each test configuration, the system writes the new result into the JSON file, adds the new result to the existing ones and immediately closes the write stream. This approach allows to save intermediate results and avoid losing hours of computation due to a crash of the system or a forced interruption of the process.

6.5 Experiments and Results

Scalability has been evaluated through three different metrics: UCC Scalability, SS Scalability, Linear Scalability. Three different test configurations have been used for each of the metrics to measure how the constraint stiffness affects the performances.

Baseline: (`!withPort`, `!withIsolation`). Uses wildcards for ports and no isolation property; represents a baseline without ports and without isolation properties.

Constrained Isolation: (`!withPort`, `withIsolation`, Bound=100). Adds a fixed load of 100 negative constraints (Isolation) to evaluate how the solver manages conflicting requirements without creating a combinatorial explosion.

Strict Semantics: (`withPort`, `!withIsolation`). Specifies strict equality on transport layer ports (TCP / 20000); significantly increases the search space of the SMT solver compared to wildcard usage.

6.5.1 UCC Scalability

The first study concerned the scaling of the number of Utility Control Center (UCC) while maintaining a fixed number of Substation (SS). Initial results showed that this metric, even though interesting in theory, provides little insight to the limitations of the computational resources of the framework. When a new UCC is added without related SS in Texas 2000 topology, a "topological island" is formed, i.e. a sub-graph with internal rules but few connections with the rest of the network. As a consequence, the solver has always solved these examples in very short times (of the order of seconds), since the complexity does not grows exponentially with the number of nodes. This case study has been considered less representative of the real-world network evolution, since the growth of control centers is generally accompanied by the creation of additional field resources. Therefore, the focus has moved to other, more complex studies.

6.5.2 SS Scalability

In this study, the number of UCCs has remained constant and the number of SSs has been incremented progressively. Even if the experimental campaigns have involved test cases with 3, 4 and 5 UCCs, the following analysis concerns only the configuration with 5 fixed UCCs because this case forces the solver to find the maximum connection density toward the same control points. The general resolution time trend for this study is illustrated in Figure 6.3 at the end of the Chapter.

Port Impact: The configuration `withPort=true` has proven to be the most computationally expensive. Indeed, in the scenarios involving 130 nodes, 22 firewalls and 72 properties, the execution times reached many hours. This shows that even if VEREFOO can handle the strict Layer 4 constraints, abstraction using wildcards is more suitable for fast prototyping of large-scale networks.

Analysis of Isolation Bound (Limit to Isolation): The configuration with `IsolationBound=100` was essential to demonstrate the validity of the negative constraints

without producing a combinatorial explosion of the isolation policies. It is an $O(N^2)$ problem to compute the isolation for each possible non-communicating pair of nodes and produce a number of constraints much larger than the one that could be reasonably produced by a realistic operation. By limiting this set to 100 significant properties, we found that the framework maintains a performance similar to that of the Baseline configuration (without Ports, without Isolation policy), with a constant delay in time. This proves that the bottleneck is not the presence of the isolation rules, but rather their potential quantity and the combinatorial complexity. The solver is able to manage effectively a realistic number of negative constraints.

6.5.3 Linear Scalability

Finally, the last and most important test case simulates the organic growth of the smart grid. We have therefore established a fixed ratio between control centers and substations (first 1:2 and then 1:3) and we have scaled up the system multiplier. This test case scales simultaneously both UCCs and SSs, as shown in Figure 6.4 at the end of the Chapter.

Convergence of Overheads due to Isolation: An interesting observation is noted in 1:3 ratio tests comparing the Baseline curve (without ports, without isolation) with the curve with Isolation Bound = 100. At first sight, for small topologies, the two curves are clearly distinct, with the curve of the configuration with Isolation showing a clear overhead. However, as the size of the topology and the multiplier increase, the curves tend to converge.

This behavior can be explained by the amortization of the complexity of the search space of the solver. Indeed, the computational effort is driven by two types of costs: the cost of satisfying the topological constraints (reachability, allocation, depending on the network size) and the cost of checking the isolation constraints (fixed at 100). In small-scale instances, the cost of checking 100 negative constraints is relatively high compared to the total resolution time. On the contrary, in large-scale instances, the solver's execution is essentially determined by the combinatorial complexity of establishing valid connectivity paths in a huge topology. As a result, the relative influence of the fixed overhead due to the isolation constraints tends to become negligible, explaining the convergence of the performance curves.

6.6 Future Implementations

As part of the experimental validation carried out in this thesis, several infrastructural issues have been identified and solved. The main issue that had to be directly treated on the source code of VEREFOO to prevent the Z3 native memory leaks (see Section 6.3.3) and the Randomized Restart strategy that was implemented in this work only at the test harness level to mitigate the solver's heuristics' stalls caused by the generation of IP addresses (Section 6.3.1) are typical examples.

Therefore, for future development of the project, it is recommended to include the resilience logic inside the framework itself, so that the caller is not required to implement the external control mechanisms, and to transform the restart strategy from an experimental workaround to a native architectural feature.

To this aim, the introduction of a new architectural element named `AdaptiveVerifier`, designed to function as an intelligent proxy between the end-user and the `VerefooSerializer` verification engine is proposed as a guideline for future implementations of the framework. The `AdaptiveVerifier` should respect the following design principles:

1. **Transparency and Encapsulation**: The `AdaptiveVerifier` should expose the same interface as the current serializer (input `NFV` object and configuration parameter `timeoutThreshold`). The main role of the `AdaptiveVerifier` is to orchestrate the execution, monitor the response times and decide autonomously whether and when to use mitigation strategies, hiding entirely the complexity to the user.
2. **Phase 1: Monitored Execution (Watchdog)**: The component starts the first verification instance on the original topology, sets the watchdog timer according to the provided threshold.
   - If the solver converges before the expiration of the timer, the result is immediately returned.
   - If the timer expires, indicating that the Z3 heuristic has stalled or explored inefficiently, the `AdaptiveVerifier` interrupts the current execution and activates the recovery process.
3. **Phase 2: "IP Permutation" Engine (Pseudonymization)**: To allow the solver to operate without modifying the semantic of the network, an internal IP Shuffling logic is suggested to be implemented:
   - **Mapping**: The system generates a temporary mapping of the original IP addresses (which cause stall) to a set of "dummy" or permuted addresses, and stores the original associations in a lookup data structure.
   - **Resolution**: The problem is submitted again to the solver using the topology with permuted IPs. The permutation of the IP addresses modifies internally the representation and the ordering of variables of the solver, causing it to adopt a different branch of the search tree.
   - **Reverse Binding**: Once a SAT result is obtained, the component uses the lookup mapping to perform the reverse binding, converting the dummy addresses present in the generated solution to the corresponding original addresses.

The introduction of this new architectural element would ensure the intrinsic robustness of the VEREFOO framework, allowing the framework to dynamically adapt to the complexity of the instances handled and providing the results in deterministic times even in the presence of "unfortunate" configurations of the underlying SMT engine.

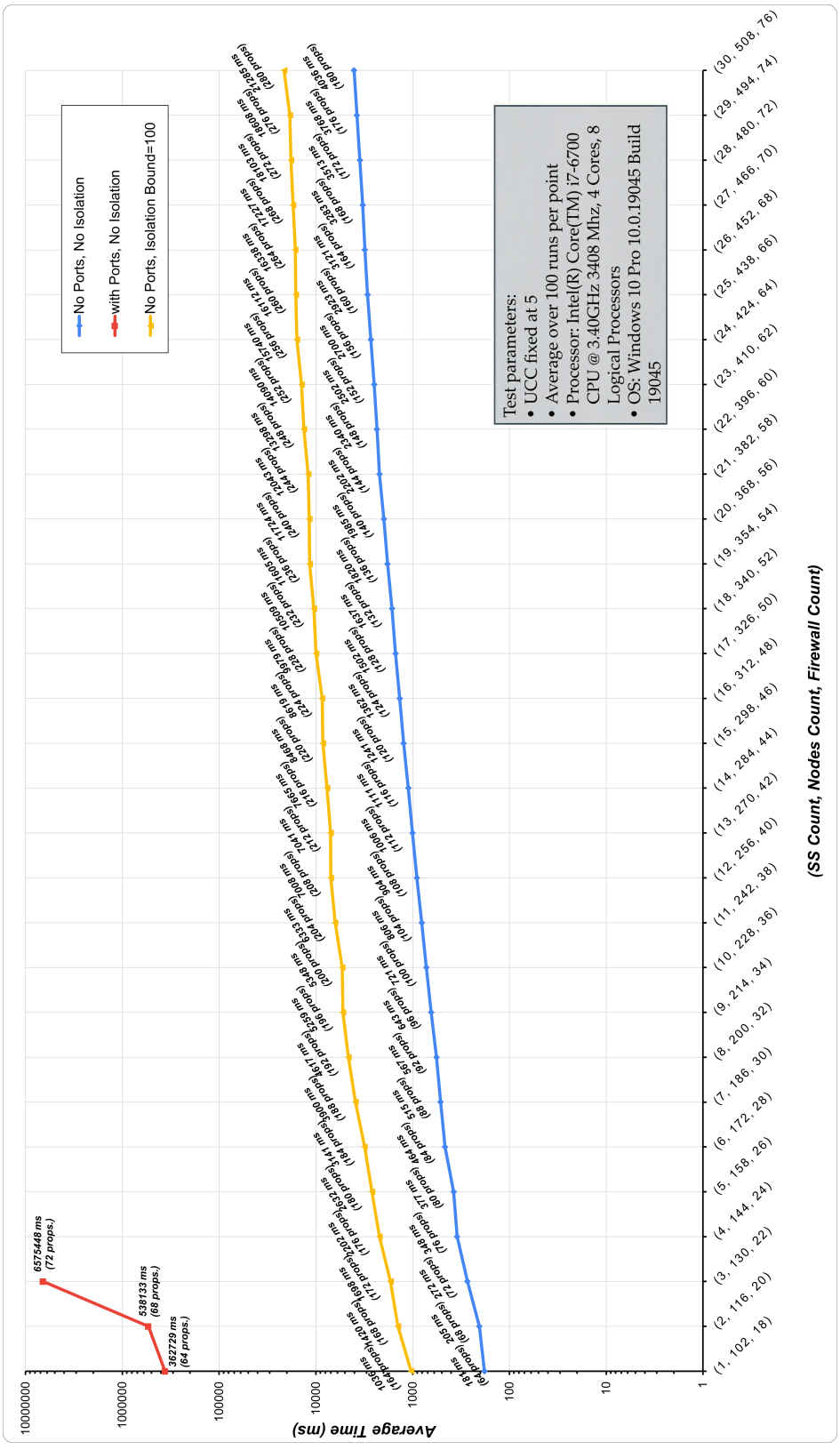


Figure 6.3: SS Scalability Test with UCC = 5

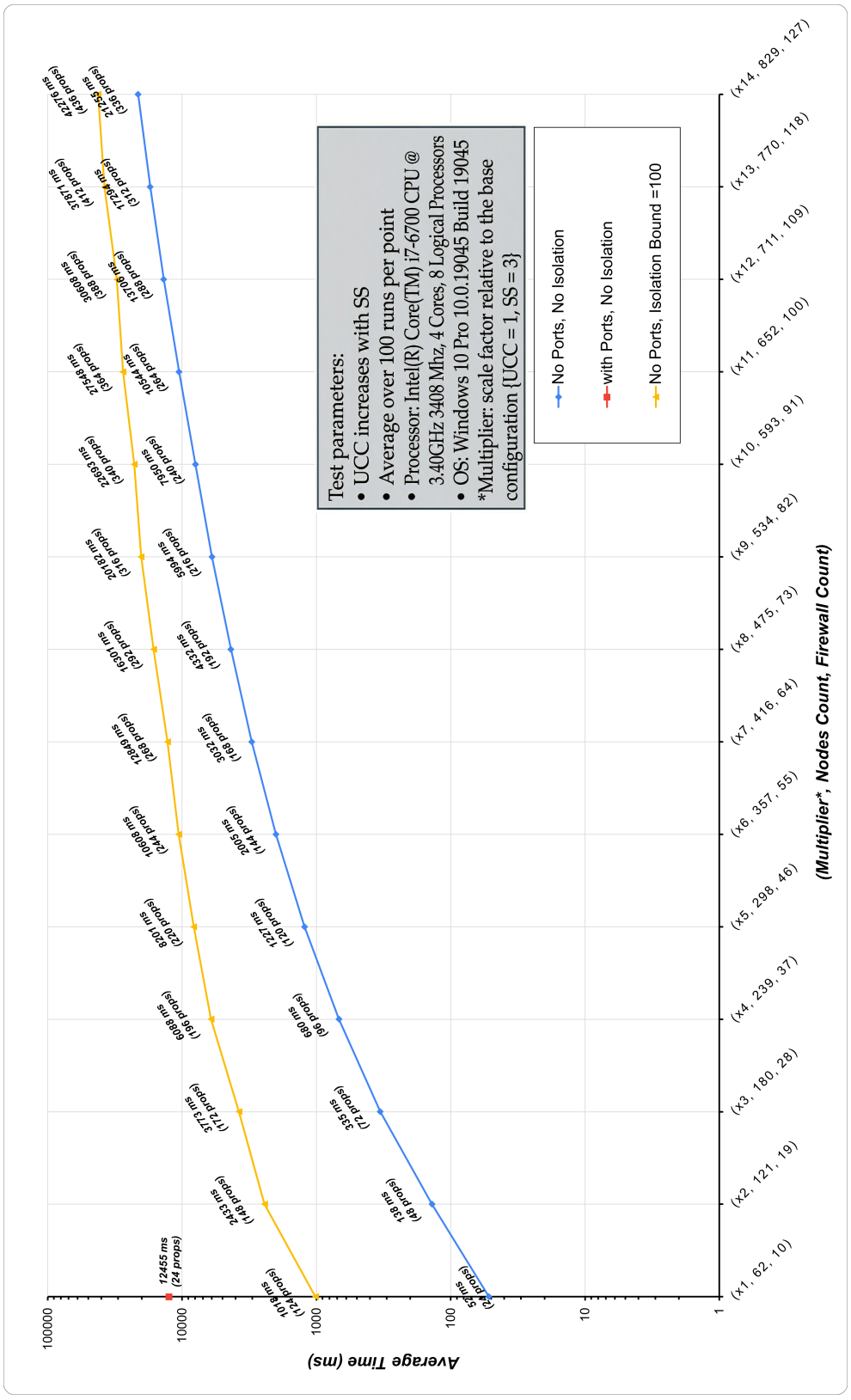


Figure 6.4: Linear scalability test with a 1:3 ratio between UCC and SS

Chapter 7

Conclusions

As part of the VEREFOO research project, the primary goal of this dissertation has been to identify and analyze the key issues in relation to the automation of network security (in SDN / NFV virtualized networks) and the integration and verification of stateful firewalls inside the VEREFOO architecture. As a consequence of the increasing demand to overcome the constraints of manual and stateless configuration, and due to the high complexity and security demands of modern Critical Infrastructures, the research adopted a formal method, based on MaxSMT problems, to demonstrate how "correct-by-construction" configurations can be guaranteed, avoiding the risk of human errors in connection tracking.

Two main contributions have emerged as a result of the completion of this research. Firstly, the objective of validating the **logical correctness (Objective 1)** of the extended stateful model, by using a rigorous cross-validation methodology, as presented in Chapter 4, to confirm that the inclusion of stateful semantics – mainly the `ALLOW_COND` action and the `cond_permit` predicate – enables the development of configurations devoid of logical ambiguity, was achieved. Moreover, tests carried out in critical scenarios showed that the proposed formal model is able to satisfy Network Security Requirements (NSRs) and avoid the risk of interference between states and vulnerabilities.

Secondly, the contribution concerning the validation of the **scalability and performance (Objective 2)** of the framework in realistic large-scale scenarios, in comparison to other theoretical approaches previously published, is significant. The thesis presents a testbed that implements the **Texas 2000** model, representing a cyber-physical infrastructure simulating a real Smart Grid, compliant with the *NERC-CIP-005-7* standards. Moreover, thanks to the implementation of a parametric test case generator (Chapter 6), the framework has been tested under several and complex topologies, characterized by hierarchical segmentation (UCC, Substations, multiple DMZs), as well as bidirectional industrial communication protocols such as DNP3 and ICCP.

Moreover, the experimental results obtained during the execution of these test cases, did not only show the computational viability ("*viability*") of the framework for large-scale networks but also helped in resolving engineering criticalities for system stability:

A memory leak native problem, occurring when interacting with the Z3 library, was identified and solved by implementing explicit resource management (`AutoCloseable`) to

ensure framework stability during prolonged executions.

The heuristic instability of the solver was reduced by implementing a Randomized Restart strategy, so that resolution times reflect the actual complexity of the problem rather than random events caused by internal solver heuristics.

Finally, the results of this thesis validated VEREFOO as a reliable solution for the orchestration of stateful firewalls, showing that the overhead generated by conditional logic is manageable, even in mission-critical scenarios.

Regarding future developments, it is suggested to internalize the resiliency strategies developed in this thesis. Specifically, the randomized restart logic currently managed by the experimental validation infrastructure, should become a native architectural component, called `AdaptiveVerifier` (technically analyzed in Section 6.6). The `AdaptiveVerifier` will act as an intelligent proxy, monitoring timeouts and autonomously applying IP shuffling techniques to unlock solver heuristic stalls, allowing the framework to be fully autonomous and deterministic in production environments.

Bibliography

[1] Ieee standard for electric power systems communications-distributed network protocol (dnp3), 2012.

[2] Adam B Birchfield, Ti Xu, Kathleen M Gegner, Komal S Shetye, and Thomas J Overbye. Grid structural characteristics as validation criteria for synthetic networks. *IEEE Transactions on Power Systems*, 32(4):3258–3265, 2017.

[3] Daniele Bringhenti. Automatic optimized firewalls orchestration and configuration in NFV environment. Master's thesis, Politecnico di Torino, 2019. Supervisors: R. Sisto, G. Marchetto, F. Valenza, J. Yusupov.

[4] Simone Bussa. Traffic flow and network security function models. Master's thesis, Politecnico di Torino, 2021. Supervisors: R. Sisto, G. Marchetto, F. Valenza, D. Bringhenti.

[5] Riccardo Sisto Fulvio Valenza Daniele Bringhenti, Simone Bussa. A two-fold traffic flow model for network security management. *IEEE Transactions on Network and Service Management*, 21(4):3740–3758, August 2024.

[6] Riccardo Sisto Fulvio Valenza Jalolliddin Yusupov Daniele Bringhenti, Guido Marchetto. Automated optimal firewall orchestration and configuration in virtualized networks. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2020.

[7] Riccardo Sisto Fulvio Valenza Jalolliddin Yusupov Daniele Bringhenti, Guido Marchetto. Automated firewall configuration in virtual networks. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1559–1576, March-April 2023.

[8] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, Cambridge, UK, 2nd edition, 2017. Focus on collision probability in hashing and random generation.

[9] Ana E Giulart Edmond Rogers Kate Davis Nastassja Gaudet, Abhijeet Sahu. Firewall configuration and path analysis for smart grid networks. In *2016 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pages 1–5. IEEE, 2016. Extracted from uploaded file: Firewall_Configuration...

[10] Brooke Nodeland. Ensuring the cybersecurity of texas' critical infrastructures. White paper, Sam Houston State University, 2023.

[11] North American Electric Reliability Corporation. CIP-005-5 — Cyber Security — Electronic Security Perimeter(s). Reliability standard, North American Electric Reliability Corporation (NERC), 2013. [Online].

[12] North American Electric Reliability Corporation. CIP-005-7 — Cyber Security — Electronic Security Perimeter(s). Reliability standard, North American Electric Reliability Corporation (NERC), Atlanta, GA, 2022. Effective Date: 01/10/2022.

[13] Christian Veloz Abhijeet Sahu Hao Huang Ana Goulart Katherine Davis Saman Zounouz Patrick Wlazlo, Kevin Price. A cyber topology model for the texas 2000 synthetic electric power grid. In *Principles, Systems and Applications of IP Telecommunications (IPTComm)*. IEEE, October 2019. Accepted for publication.

[14] Luana Pulignano. Security automation for stateful firewalls. Master degree thesis, Politecnico di Torino, Torino, Italy, 2024. Supervisors: prof. Fulvio Valenza, prof. Riccardo Sisto, dott. Daniele Bringhenti. Academic Year 2023-2024.

[15] Karen Scarfone and Paul Hoffman. Guidelines on firewalls and firewall policy. Technical Report Special Publication 800-41 Revision 1, National Institute of Standards and Technology, Gaithersburg, MD, 2009.

[16] Keith Stouffer, Victoria Pillitteri, Suzanne Lightman, Marshall Abrams, and Adam Hahn. Guide to industrial control systems (ics) security. Technical Report Special Publication 800-82 Revision 2, National Institute of Standards and Technology, 2015.