Politecnico di Torino

Master of Science in Cybersecurity

Master Degree Thesis

A Parallelization Approach Via Clusterization for Firewall Configuration

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

Candidate
Giuseppe Salvemini

Academic Year 2024-2025

Summary

The growing complexity and dynamic nature of modern computer networks pose significant challenges for ensuring their security and correctness. Network misconfigurations are the most common reason for vulnerabilities, most times resulting in security breaches, disruptions of service, as well as non-compliance. Formal verification, a rigorous methodology rooted in mathematical logic, offers the systematic method to understand network configurations and give robust guarantees for their actions and adherence to the security policies. One of the most visible tools in this area is VEREFOO, designed at the Politecnico di Torino, excels for its capacity to efficiently fulfill network needs by allocating firewalls to avoid subtle configuration errors. While VEREFOO has proven highly successful in evading critical bugs, its use on large-scale, enterprise-level networks frequently incur severe performance bottlenecks. Computational cost needed to exhaustively search the entire state space of complex network topologies grows exponentially, leading to unfeasibly long analysis times and memory consumption. This inherent scalability limitation restricts VEREFOO's practical use for eventual real-world deployments where quick verification loops are repeatedly iterated frequently essential for maintaining a strong security posture. This master's thesis addresses the crucial problem of VEREFOO's scalability by proposing and assessing a parallelization strategy using network clusterization. The key idea involves intelligently partitioning a large, monolithic network into a set of smaller, more manageable subnetworks. These subnetworks may then be analyzed concurrently by separate instances of VEREFOO. This concurrent processing paradigm leverages modern multi-core architectures, promising a substantial reductions in aggregate verification time. After this phase, an aggregation algorithm is used to collect the individual outputs of verification and aggregate them in a final solution. The key benefit of this approach is its high capability to increase the configuration speed significantly and to let VEREFOO process high scale networks, thus allowing formal verification becoming increasingly accessible and embedded feature of regular security assurance processes. However, this performance enhancement comes with an acknowledged trade-off: the potential loss of global optimality in the verification solution. By analyzing subnetworks in isolation, certain inter-subnetwork dependencies or global properties that span across partition boundaries might not be fully captured or optimized, leading to a "good enough" rather than a perfectly optimal solution. Despite this trade-off, the cybersecurity implications are profound. In dynamic environments where frequent integration and quick deployment are essential, a faster, larger scale verification procedure with highly accurate, potentially non-optimum, results are frequently more beneficial than an optimum solution delivered too late. This thesis fills the

gap between theoretical rigor and usability for practical network security verification, opening the door to wider and better use of formal methods when securing the critical network infrastructures against evolving cyber threats.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis objective

Modern computer networks form the backbone of global commerce, communication and critical infrastructure: from massive cloud installations to advanced corporate intranets to highly interconnected IoT ecosystems. Their scope, size, and heterogeneity continue to expand, this rapid development, while permitting unprecedented connectivity and innovation, is introducing numerous security issues. Network configuarations, which establish who can access what and how security policies are enforced, have become highly complex. One misconfiguration, both accidental and intentional, can result in great weaknesses, which can lead to data breaches, denial-of-service attacks and severe financial and reputational risk. Traditionally, network security assurance has been highly reliant on manual configuration, which by nature is limited, indeed as stated in [1] and [2] manual procedures are prone to human error, labor-intensive, and incapable of sustaining pace with the explosive development of large networks. The sheer volume of security policies and the very massive scale of current networks make it impractical for human basic rule-based systems or experts to rigorously verify network correctness and security policy compliance. This case illustrates the necessity of increased systematic, automated, and rigorous means of checking network configurations. Here, the formal verification is one dominant paradigm to counter these weaknesses, which has a foundation in mathematical logic and computer science. Formal verification techniques aim to prove or disprove the correctness of a system with respect to a formal specification. When applied to networks, this consists in creating a mathematical model of the network's configuration and behaviour, defining desired security properties and then using automated tools to formally verify if the network model satisfies these properties. This approach offers a level of rigor and completeness not feasible for traditional manual methods, capable of uncovering subtle, non-obvious errors that could otherwise be exploited.

Among all the state-of-the-art tools concerning the network formal verification frontier, VEREFOO stands out prominently. Developed by the Politecnico di Torino, VEREFOO has been made to provide automatic, precise, and complete verification of network configuration. The strength of VEREFOO lies in the rigid application

of formal methods, often leveraging methods such as Satisfiability Modulo Theories (SMT) solvers, model checking, or symbolic execution. By translating the network topology and security properties into the formal logical representation, VEREFOO can examine every one of the potential interactions and the traffic paths thoroughly, generating definitive answers regarding the network properties. It allows network admins as well as security engineers to be highly sure regarding the network security state, detect issues ahead of time and fix them before deployment, and be constantly compliant. Despite the clear strengths of the formal verification tools like VEREFOO, the real-world constraint is scalability. As the network size grows, computational resources that have to be invested in exhaustive formal verification scale exponentially. This limitation translates to the fact that VEREFOO performs very well on smaller static networks or on subtrees of high critical nature, but it very soon fades when it is matched against the scale and dynamism of the modern-day enterprise, cloud, or service provider network. Getting above the scalability barrier is the answer that can unlock the full potentials of formal network verification and make it a necessity of modern cybersecurity practice.

This thesis directly addresses the issue of scalability faced by VEREFOO. The primary goal of the current work is to significantly enhance the speed and scalability of VEREFOO, and consequently, its application's extendability to large and complex network configurations that currently fall beyond its practical capability. To achieve this, it is proposed and analyze a new parallelization scheme using the network decomposition principle. The entire methodology consists in cleverly dividing a massive, highly interconnected network into a collection of smaller, more manageable subnetworks. The decomposition is such that it reduces inter-subnetwork dependencies without altering the overall essence of the original network in each partition. After the decomposition, each subnetwork will be studied simultaneously and independently by separate instances of VEREFOO. This simultaneous processing paradigm will yield significant performance improvements by distributing the computational load across multiple processing cores or distributed computing nodes. By subdividing a solitary, intractable problem into many small, manageable problems, we aim to significantly alleviate the overall verification time. After conducting parallel analysis of individual subnetworks, the results retrieved from every occurrence of VEREFOO are then aggregated and incorporated to provide a complete verification analysis for the entire original network. This aggregation process should be crafted thoughtfully such that the pooled results do actually reflect the network state globally. Though the provided parallelization and decomposition methodology offers substantial improvements in speed and scalability, it's critical to appreciate an intrinsic trade-off: the likely deviation from global optimality of the solution. After the decomposition into isolated subnetworks, particular global features or complicated relations that persist throughout the boundaries of such partitions might not necessarily be accurately documented or strictly verifiable by the VEREFOO instances independently. For instance, a security policy that dictates flow of traffic over various distinct subnetworks may be hard to check best when subnetworks in isolation are considered. The local analysis might yield an optimal result for that specific subnetwork, but the overall results might not give the ultimate 'most optimal', 'most complete', verification for the entire network.

This thesis aims to contribute to the improvement of automated network security by providing an efficient and practical method of scaling formal verification. By bringing together the pursuit of optimal solutions and the need for speed and scalability, the thesis attempts at making rigorous network configuration assurance an integral component of today's cybersecurity practices. The following chapters will discuss about the methodology, implementational aspects, experimental analysis, and thoughtful consideration of the involved trade-offs of this parallel decomposition approach.

1.2 Thesis description

The remaining chapters of this thesis are organized as follows:

- **Chapter 2:** This chapter presents the way in which network packets are modeled within VEREFOO. It explains the logic behind representing network communication at the packet level and introduces two key strategies for grouping packets into traffic flows: Maximal Flows and Atomic Flows. The chapter discusses in detail how each approach captures different levels of granularity in traffic analysis and highlights the respective advantages and disadvantages of these two methodologies.

- **Chapter 3:** This chapter provides a brief yet comprehensive overview of VEREFOO, focusing on its internal architecture and functioning. It explains how both the network topology and the associated security requirements are modeled within the framework. Furthermore, it introduces the MaxSMT problem, the core verification mechanism that enables the automatic configuration of network security devices, while also discussing why this process becomes computationally expensive and therefore unsuitable for large-scale networks.

- **Chapter 4:** This chapter outlines the primary objectives of the thesis and provides a summary of the methodological approach adopted to achieve them. It defines the research questions that guided the work and offers an overview of how the proposed solution was structured to enhance VEREFOO's scalability and efficiency.

- **Chapter 5:** This chapter presents an in-depth description of the scalability approach developed during this research. It details the use of the Leiden algorithm for clustering the network into smaller, manageable subnetworks, as well as the strategy for handling inter-cluster dependencies. In addition, it describes the merge algorithm, which combines the results obtained from each cluster to reconstruct a coherent overall configuration, ensuring both correctness and completeness of the final solution.

- **Chapter 6:** This chapter focuses on the validation and performance evaluation of the proposed approach. It presents a series of experiments designed to assess how the parameters of the Leiden algorithm—such as resolution and

minimum node size—affect the resulting clustering. The chapter also compares the performance and scalability of the new approach with the standard version of VEREFOO, demonstrating the improvements achieved in terms of computation time and verification efficiency.

- **Chapter 7:** The final chapter summarizes the conclusions of the thesis. It revisits the goals established at the beginning of the work, evaluates the extent to which they were successfully achieved, and discusses potential directions for future improvements and research. This includes possible optimizations of the current implementation, extensions to other types of network configurations, and broader applications of the developed methodology.

Chapter 2

Traffic Flows

To effectively analyze and verify the security properties of a computer network, tools like VEREFOO cannot directly interact with the physical network infrastructure. Instead, they operate on abstract, mathematical representations of the network and its behavior. These representations are known as formal models. The use of formal models is fundamental to formal verification because they provide a precise, unambiguous, and rigorous framework for describing complex systems. Without such models, it would be impossible to apply mathematical logic and automated reasoning to guarantee the correctness or identify flaws in network configurations. The process of formal verification within VEREFOO, therefore, begins with translating the real-world network into these structured models. These models must capture all the information relevant to the security properties being verified, such as how data travels, how network devices process that data, and what security rules are in place. At the same time, these models must be designed efficiently enough to allow for practical computation, balancing the need for accuracy with the demands of computational tractability. If a model is too simplistic, it might miss critical details leading to inaccurate verification results. Conversely, if it is overly detailed, the verification process could become computationally infeasible, especially for large and dynamic networks. For VEREFOO to perform its verification tasks, it relies on several interconnected formal models, each representing a different aspect of the network security problem. These include models for the network itself, for the security requirements that need to be enforced, and crucially, for the packets that traverse the network, which then form the basis for traffic flows. In this chapter will be briefly described the network model and the NSR model but they will be analyzed deeper in later, while it will focus on the traffic flows.

2.1 The Network Model

The network model is essentially a blueprint of the network's structure and the behavior of its components. It represents the network's topology, which includes all the devices (nodes) and the connections between them (links). Nodes can represent various network elements, such as client computers, servers, routers, firewalls, Network Address Translators (NATs), and load balancers. Beyond just the physical layout, the network model also formally describes the behavior of each network

function (NF). This is critical because network devices don't just forward packets; they can also modify them (e.g., NATs changing IP addresses) or decide to drop them based on specific rules (e.g., firewalls). VEREFOO needs to understand precisely how each of these functions processes incoming packets and what happens to them as they traverse the device. This includes modeling their forwarding behavior (which packets are allowed to pass and which are discarded) and their transformation behavior (how packet attributes are changed). We will delve deeper into the specifics of the network model in a later section, as it forms a comprehensive representation of the network's operational logic.

2.2 The Network Security Requirements Model

The Network Security Requirements (NSR) model defines the desired security policies and behaviours that the network is expected to uphold. These are the "specifications" against which VEREFOO verifies the network's actual configuration. NSRs typically express connectivity policies, specifying which types of traffic are allowed to reach certain destinations (known as reachability policies) and which types of traffic must be prevented from reaching specific destinations (known as isolation policies). These policies are formally expressed as conditions on packet attributes and actions (allow or deny). The NSR model provides the ground truth for verification, allowing VEREFOO to determine if the network's configuration correctly implements these critical security directives. The detailed structure and formalization of NSRs will also be discussed in a subsequent section.

2.3 The Traffic Model

To provide a concrete foundation for VEREFOO's operations, the packet class model precisely defines traffic, t, as a predicate over the values of the TCP/IP 5-tuple packet fields. Specifically, t is represented as a disjunction of predicates, $q_{t,1} \vee q_{t,2} \vee \cdots \vee q_{t,n}$, where each $q_{t,i}$ is a conjunction of five predicates, one for each field of the 5-tuple: source IP address (IPSrc), destination IP address (IPDst), source port (pSrc), destination port (pDst), and transport protocol (tPrt). For IPv4 addresses, IPSrc and IPDst are conjunctions of four predicates, one for each byte of the IP address, allowing for single integer values or ranges (e.g., 130.192.5.* identifies addresses matching 130.192.5.0/24). Similarly, pSrc and pDst can identify single integer port numbers or ranges (e.g., 80 or [80, 100]), while tPrt can specify single or subsets of values (e.g., "TCP" or "UDP"). The wildcard * concisely represents the full range of values for a field. Furthermore, the concept of a "sub-traffic" ($t1 \subseteq t2$) is defined, indicating that t1 represents a subset of packets represented by t2, which is crucial for understanding flow relationships.

2.4 The Packet Model and the Genesis of Traffic Flows

At the most granular level, network communication is composed of individual packets. Each packet carries specific information in its header, such as source IP address, destination IP address, source port, destination port, and the transport protocol (often referred to as the IP 5-tuple). For VEREFOO to analyze network behavior, it must first understand these fundamental units of data. The packet class model is designed to identify and group packets that share common characteristics and are expected to be treated similarly by network functions. Instead of modeling every single packet individually (which would be computationally impossible given the sheer number of possibilities), VEREFOO groups them into packet classes. A packet class is formally defined as a predicate over the values of packet fields. For instance, a packet class might be defined as "all TCP packets originating from IP address 192.168.1.100 destined for port 80 on any IP address." This abstraction allows VEREFOO to reason about the behavior of entire groups of packets rather than individual ones. This concept of grouping packets naturally leads to the notion of traffic flows. A traffic flow, in the context of VEREFOO, represents the journey of a specific packet class through the network. It describes not only the path a packet class takes from its source to its destination but also how its attributes might be transformed by intermediate network functions along that path. A traffic flow f is formally modeled as a list of alternating nodes and predicates $[n_s, t_{sa}, n_a, t_{ab}, n_b, ..., n_k, t_{kd}, n_d]$. Each node in the list corresponds to a node crossed by the flow in the path, starting from the source node n_s (that generates traffic t_{sa}) and arriving at the destination node n_d (that receives traffic t_{kd}). This list shows how a packet class is transmitted between nodes and how it might be transformed or dropped. This approach is highly advantageous because it allows the verification algorithms to operate on a manageable number of equivalent classes of packets, rather than dealing with the countless individual packets that could traverse the network. The way these packet classes are grouped into traffic flows is a critical design choice, as it directly impacts the computational efficiency and the level of detail in the verification process. Different grouping strategies lead to different trade-offs between the number of flows that need to be analyzed and the granularity (or specificity) of each flow. This section will now focus extensively on these different strategies for defining and computing traffic flows, specifically examining Atomic Flows and Maximal Flows.

2.4.1 Traffic Flow Aggregation Strategies: Atomic Flows vs. Maximal Flows

While a network can theoretically be crossed by an infinite number of distinct traffic flows, formal verification tools must focus on a relevant subset. The key challenge lies in how to aggregate individual packets into meaningful flow entities that are both representative of network behavior and computationally tractable. The paper "A Two-Fold Traffic Flow Model for Network Security Management" [3] explores two contrasting strategies for this aggregation: Atomic Flows and Maximal

Flows. These two approaches represent different philosophies regarding the balance between the number of flows generated and their granularity, each with distinct implications for performance and verification accuracy.

2.4.2 Atomic Flows: Precision through Disjoint Minimality

The concept of Atomic Flows is inspired by "Atomic Predicates," a powerful idea from prior research in network verification. At its core, the Atomic Flow strategy aims to identify the smallest possible, mutually disjoint classes of packets that are treated identically by every network function (like filters and transformers) within the network. Imagine you have a set of rules, each defined by a condition on packet fields (e.g., "source IP is 10.0.0.1" or "destination port is 80"). These conditions can overlap. An Atomic Predicate is a unique, minimal combination of these conditions that doesn't overlap with any other Atomic Predicate. For instance, if you have a rule for 10.0.0.0/24 and another for 10.0.0.1, the Atomic Predicates would be 10.0.0.1 and 10.0.0.0/24 AND NOT 10.0.0.1. Every packet in the network will belong to exactly one Atomic Predicate. When this concept is extended to traffic flows, an Atomic Flow is defined as a flow where every packet class within it corresponds to an Atomic Predicate. This means that each Atomic Flow represents a very specific and minimal group of packets, and crucially, no two Atomic Flows will ever represent the exact same set of packets at any point in the network. **Advantages of Atomic Flows:**

- **Fine Granularity:** Because each packet class in an Atomic Flow is minimal and disjoint, the verification can be extremely precise. Every nuance of how different packet types are handled is explicitly represented.
- **Simplified Representation:** A significant benefit is that each unique Atomic Predicate can be assigned a unique integer identifier. This allows VEREFOO's internal computations to operate on simple integer numbers instead of complex logical expressions (like conjunctions of predicates over multiple packet fields). This simplification can lead to more efficient processing in subsequent verification steps, particularly when dealing with constraint-based programming problems like Satisfiability Modulo Theories (SMT).
- **Clearer Disjointness:** The inherent disjointness of Atomic Predicates simplifies reasoning about traffic paths and transformations, as there's no ambiguity about which flow a packet belongs to.

Disadvantages of Atomic Flows:

- **Large Number of Flows:** The pursuit of minimal and disjoint packet classes often results in a significantly larger number of Atomic Predicates, and consequently, a much larger number of Atomic Flows. This increased number of entities can introduce an overhead in the initial computation of these flows and potentially in the memory required to store them.
- **Initial Computation Cost:** The process of identifying all Atomic Predicates across an entire network, considering all possible intersections and transformations, can be computationally intensive as an initial step.

2.4.3 Maximal Flows: Efficiency through Aggregation

In contrast to Atomic Flows, the Maximal Flows strategy operates on an opposite principle: it aims to maximize the aggregation of traffic flows. The goal is to reduce the total number of distinct flows that VEREFOO needs to consider by grouping together all packet classes that behave identically as they traverse the network. A Maximal Flow aggregates all subflows that follow the same path and undergo the same sequence of transformations and forwarding decisions. If multiple packet classes, even if distinct at their origin, end up being processed identically by every network function along a particular path, they are grouped into a single Maximal Flow. This means a single Maximal Flow can represent a much broader class of packets compared to an Atomic Flow. **Advantages of Maximal Flows:**

- Reduced Number of Flows: The most immediate benefit is a substantial reduction in the total number of traffic flows that need to be computed and processed. This can lead to faster initial flow computation times, as there are fewer entities to track.
- Potentially Lower Memory Usage (for flow representation): With fewer flows, the memory required to store the flow entities themselves might be lower, especially if the complexity of individual packet class predicates is not excessively high.

Disadvantages of Maximal Flows:

- **Coarser Granularity:** While reducing the number of flows, Maximal Flows inherently represent broader, less specific packet classes. This means that a single Maximal Flow might encompass a wide range of packets, some of which might behave slightly differently in subtle ways that are not captured by the aggregation. This can lead to a less precise verification result in certain scenarios.
- **Complex Packet Class Representation:** Unlike Atomic Predicates, the packet classes within Maximal Flows are typically represented by complex logical predicates (e.g., conjunctions and disjunctions of conditions on IP 5-tuple fields). These predicates cannot be simplified to single integer identifiers. Consequently, VEREFOO's internal algorithms must work with these more complex representations, which can increase the computational burden in the later stages of verification, particularly for solvers that prefer simpler, numerical inputs.
- **Non-Disjoint Packet Classes:** The packet classes represented by different Maximal Flows might not be mutually exclusive; they can overlap. This non-disjointness can complicate reasoning and require more sophisticated handling during the verification process to avoid double-counting or misinterpreting results.

Chapter 3

VEREFOO

This chapter delves into the intricacies of VEREFOO [4], the formal verification tool developed at Politecnico di Torino, which forms the core subject of this thesis. Building upon the foundational concepts of formal models and traffic flow discussed in the preceding chapter, we will now explore VEREFOO's architecture and operational mechanisms in greater detail. Understanding these specifics is crucial for appreciating the challenges of scalability that this thesis aims to address through parallel decomposition. We will begin by examining VEREFOO's network modeling approach, specifically its use of Service Graphs and Allocation Graphs, and how various network functions like NATs, Firewalls, and Forwarders are represented. Following this, we will explore how Network Security Requirements are formally captured within VEREFOO, providing the essential specifications for verification. Finally, we will dissect the underlying Maximum Satisfiability Modulo Theories (MaxSMT) problem, which VEREFOO leverages to achieve automated, optimal, and formally correct solutions for network security management.

3.1 The Network Model in VEREFOO: Service Graph and Allocation Graph

To effectively apply formal verification techniques, VEREFOO relies on precise and abstract representations of the network. These representations, known as formal models, serve as the foundation upon which all analysis and problem-solving are performed. VEREFOO primarily utilizes two interconnected models to describe the network: the Service Graph (SG) and the Allocation Graph (AG).

3.1.1 The Service Graph (SG)

The Service Graph (SG) provides a high-level, logical abstraction of a virtual network. It represents the interconnection of various service functions and network nodes that collectively deliver an end-to-end network service. Unlike a physical network diagram, the SG focuses on the functional relationships and traffic paths, abstracting away the underlying physical infrastructure. It is typically defined by a

network service designer whose primary concern is to provide a functional networking service to users, with security considerations often handled separately or at a later stage. Formally, an SG is modeled as a directed graph, $G_S = (N_S, L_S)$, where N_S is the set of vertices (nodes) and L_S is the set of directed edges (links) representing connections between these nodes. The set of nodes, N_S, is further divided into two disjoint subsets: E_S, representing the endpoints (such as single hosts or entire edge subnetworks like client machines or servers), and S_S, representing the middleboxes (various service functions). Each node in N_S is uniquely identified by a non-negative integer index, k, denoted as n_k. Crucially, each node $n_k \subseteq N_S$ is also characterized by a specific IP address, an IP address range, or, more generally, a set of IP addresses. Low-level functions like switches and routers, which primarily forward packets based on routing tables, are typically not explicitly included in the SG. Instead, the SG assumes that these underlying functions correctly implement the connections between the more complex service functions, providing a simplified yet functionally complete view of the network's logical paths.

3.1.2 The Allocation Graph (AG)

The Allocation Graph (AG) is an internal representation derived automatically from the user-provided SG. Its purpose is to introduce Allocation Places (APs), which are placeholder elements where VEREFOO can potentially decide to allocate a firewall instance. This transformation from SG to AG is crucial for the automated firewall allocation process. Similar to the SG, the AG is also modeled as a directed graph, $G_A = (N_A, L_A)$, maintaining the same indexing scheme for its vertices and edges. However, the set of nodes in the AG, N_A, is a union of three disjoint subsets: E_A, S_A, and A_A. Here, E_A and S_A correspond directly to the endpoints and middleboxes from the original SG ($E_A = E_S$ and $S_A = S_S$), meaning these core network components remain unchanged. The new set, A_A, comprises the Allocation Places. For each link between any pair of network nodes or functions in the original SG, a placeholder AP element can be generated in the AG. The generation of APs and the overall structure of the AG can be influenced by specific requirements provided by the security designer. These requirements are formalized using two predicates: $forbidden : L_S \rightarrow B$ and $forced : L_S \rightarrow B$. The $forbidden(l_{i,j})$ predicate is true if the creation of an AP on a specific link $l_{i,j}$ from the SG is prohibited. Conversely, $forced(l_{i,j})$ is true if the allocation of a firewall on that link is explicitly required. Based on these forbidden constraints, A_A and L_A are computed to be the smallest sets satisfying certain conditions. For instance, if an AP is not prohibited on an SG link, an AP node ah is inserted between the original nodes n_i and n_j, replacing the original link with two new links ($l_{i,h}$ and $l_{h,j}$). If an AP is prohibited, the original link $l_{i,j}$ is simply included directly in the AG's links. The forced predicate then dictates that if a firewall is explicitly required on a link, the corresponding AP ah must have an allocated firewall. A predicate $allocated : N_A \rightarrow B$ is introduced to formalize allocation decisions, specifying whether a network node (particularly an AP) is actually allocated in the AG. For existing endpoints and service functions ($n_k \subseteq E_A \cup S_A$), $allocated(n_k)$ is true by definition. For APs ($a_h \subseteq A_A$), the automated procedure decides whether $allocated(a_h)$ should be true (i.e., a firewall is placed there) or false.

3.1.3 Network Functions (NFs): NAT, Firewalls, and Forwarders

The behavior of each Network Function (NF) within the network, whether it's a pre-existing middlebox in the SG/AG or a newly allocated firewall in an AP, is abstractly modeled in VEREFOO using two key functions:

- **Forwarding Behavior:** This is captured by a predicate $deny : T \rightarrow B$, where T represents a packet class and B is the Boolean set (true, false). $deny(t)$ is true if the NF drops all packets belonging to the input packet class t due to its configuration. This function helps determine which traffic is permitted or blocked.
- **Transformation Behavior:** This is modeled by a function $T_i : T \rightarrow T$, called a transformer. It maps an input packet class to the corresponding output packet class after the NF has processed it.

It's important to note that these two functions are kept distinct and independent in VEREFOO's model. A transformer describes packet modifications regardless of whether the packet is ultimately dropped. This separation allows for a more flexible and modular approach to modeling network functions. Let's examine how specific types of NFs are modeled:

- **Forwarder:** This is the simplest type of network function. For a forwarder, the transformation function T_i is the identity function, meaning it does not modify the packets it processes ($T_i(t) = t$). Additionally, its deny predicate is typically false for all traffic, as its primary role is simply to forward packets.
- **Firewall:** Firewalls are crucial for enforcing security policies by filtering traffic. In VEREFOO's model, a firewall is characterized by an identity transformation function ($T_i(t) = t$), as firewalls do not typically modify packet headers (unlike NATs). Its core behavior is encapsulated in the $deny_i(t)$ predicate, which is true if packets of class t are dropped according to the firewall's configured rules. The complexity of a firewall lies entirely in its filtering logic, which determines which packets are allowed to pass and which are blocked based on their attributes. Note that at the moment only packet filters are supported by VEREFOO, which are the most common type of firewalls.
- **NAT (Network Address Translator):** NATs are examples of network functions with non-identity transformation behaviors, as they modify packet headers, typically IP addresses and/or port numbers. VEREFOO models a NAT's behavior based on the type of transformation it performs. For instance, a common NAT might perform address translation, where source addresses are translated into public addresses (shadowing) or destination addresses are reconverted into shadowed addresses. If no such conditions are met, the packet might remain unmodified. This complex behavior is expressed as a disjunction of transformations: $T_i(t) = \vee_j(T_{i,j}(D_i, j \wedge t))$. Here, $D_{i,j}$ represents the

specific packet class that triggers a particular transformation $T_{i,j}$. For example, if p_x represents a shadowed IP address and a_y a public IP address, the model defines specific predicates like $D_{i,1}$ (source is shadowed, destination is not) for shadowing, $D_{i,2}$ (source is not shadowed, destination is public) for reconversion, and $D_{i,3}$ (all other cases) for identity transformation. Each $T_{i,j}$ then describes how the packet fields are modified (e.g., IPSrc changing to a_y for shadowing). This detailed modeling allows VEREFOO to accurately track how traffic is altered as it traverses NAT devices, which is fundamental for correct security verification.

These are only some of the Network Functions that can be modelled, other complex NFs exist and can be supported, for example VEREFOO can deal with VPNs as shown in [5].

3.2 Network Security Requirements (NSRs) Modeling in VEREFOO

Network security is fundamentally about enforcing specific rules and behaviors. In VEREFOO, these desired security behaviors are formally captured by Network Security Requirements (NSRs). NSRs express the high-level security policies that the network must satisfy, serving as the "ground truth" against which the configured network is verified. An NSR, denoted as r, is formally modeled as a pair $r = (C, a)$, where:

- C is a condition that defines the specific traffic (a packet class) to which the requirement applies. This condition is structured similarly to the packet class model, based on the IP 5-tuple fields (IPSrc, IPDst, pSrc, pDst, tPrt). The IPSrc and pSrc predicates specify the traffic sources, while IPDst, pDst, and tPrt specify the traffic destinations and protocol.
- a is the action that must be performed on the traffic matching condition C. The action can be either ALLOW or DENY.

Based on the action, NSRs are categorized:

- **Isolation Requirements (r.a = DENY):** These policies specify that certain traffic flows must be prohibited from reaching a particular destination. For example, "traffic from the guest network to the internal database must be denied."
- **Reachability Requirements (r.a = ALLOW):** These policies specify that certain traffic flows must be allowed to reach a particular destination. For example, "traffic from the web server to the external payment gateway must be allowed."

For a traffic flow $f = [e_s, t_{s,a}, \ldots, t_{k,d}, e_d]$ to satisfy a condition C, three conditions must be met:

1. Its source endpoint es and destination endpoint ed must have IP addresses matching C.IPSrc and C.IPDst, respectively.
2. The traffic originating from the source, $t_{s,a}$, must satisfy $C.IPSrc$ and $C.pSrc$.
3. The traffic arriving at the destination, $t_{k,d}$, must satisfy $C.IPDst$, $C.pDst$, and $C.tPrt$.

The set of all flows that satisfy a policy $p.C$ is denoted as $F_p \subseteq F$. An important property is that if a flow is in F_p, then all its subflows are also in F_p. VEREFOO supports different high-level approaches for specifying NSRs, which influence how default behaviors are handled:

- **Whitelisting:** In this approach, the default behavior is to block all traffic. Users explicitly specify only ALLOW requirements. Any traffic not explicitly allowed is implicitly denied.
- **Blacklisting:** Conversely, the default behavior is to allow all traffic. Users explicitly specify only DENY requirements. Any traffic not explicitly denied is implicitly allowed.
- **Rule-Oriented Specific:** Users can explicitly formulate both ALLOW and DENY properties. The system automatically decides how to manage other cases (traffic not covered by specific rules) with the objective of minimizing the number of generated firewall rules.
- **Security-Oriented Specific:** Similar to rule-oriented specific, users define both ALLOW and DENY properties. However, the system's objective here is to allow only the communications that are strictly necessary to satisfy all user requirements, often leading to a more restrictive default.

The specific NSRs provided by the user form the set R_s. Additionally, a set R_D is defined to represent the default behavior (if applicable for whitelisting or blacklisting). The total set of requirements considered by VEREFOO is $R = R_s \cup R_D$. Finally, the core of how VEREFOO formally checks these requirements against the network model is expressed through logical formulas:

- For an isolation requirement r (action DENY), the model ensures that for every flow f that satisfies $r.C$, there must exist at least one allocated network node n_i along that flow's path ($\pi(f)$) that drops the traffic $\tau(f, n_i)$ at its ingress. In simpler terms, to block a specific type of traffic, at least one firewall or denying function must be placed in its path to drop it.

$$\forall f \in F_r.\exists i.(n_i \in \pi(f) \wedge allocated(n_i) \wedge deny_i(\tau(f, n_i))) \tag{3.1}$$

- For a reachability requirement r (action ALLOW), the model ensures that there must exist at least one flow f that satisfies $r.C$ such that for all allocated network nodes n_i along its path, the traffic is not denied. This means that at least one valid path must exist where the traffic is allowed to pass through all intermediate allocated devices.

$$\exists f \in F_r.\forall i.(n_i \in \pi(f) \wedge allocated(n_i) \implies \neg deny_i(\tau(f, n_i)) \tag{3.2}$$

Additionally, VEREFOO also incorporates the concept of a complete reachability requirement, which, while similar to a standard reachability requirement, specifically mandates that all flows satisfying the condition $r.C$ must not be blocked by any node along their paths.

3.3 The MaxSMT Problem in VEREFOO

The central mechanism by which VEREFOO achieves its objectives of automated, optimal, and formally correct firewall allocation and configuration is by formulating and solving a partial weighted Maximum Satisfiability Modulo Theories (MaxSMT) problem.

3.3.1 Introduction to MaxSMT

MaxSMT is a powerful extension of the traditional Satisfiability Modulo Theories (SMT) problem. An SMT problem involves determining if a given set of first-order logic constraints (formulas expressed in a combination of logical and mathematical theories, such as arithmetic or bit-vectors) can be simultaneously satisfied. MaxSMT takes this a step further by introducing an optimization component. It distinguishes between two types of constraints:

- **Hard Clauses:** These are non-relaxable constraints that must be satisfied for any valid solution to exist. If even one hard clause is violated, the problem has no solution. They define the core correctness requirements of the system.
- **Soft Clauses:** These clauses are assigned a numerical weight. Their satisfaction is not strictly required, but the objective of the MaxSMT solver is to find a solution that maximizes the sum of the weights of all satisfied soft clauses. This allows for expressing optimization goals, where some conditions are preferred but not strictly mandatory

While MaxSMT problems are computationally complex (NP-complete in the worst case), modern state-of-the-art solvers, such as Z3 (used by VEREFOO), employ sophisticated algorithms and heuristics that can efficiently solve many real-world instances in polynomial time.

3.3.2 Why VEREFOO Leverages MaxSMT

The choice of MaxSMT as the underlying problem formulation is fundamental to VEREFOO's design, enabling it to achieve its three main objectives:

- **Full Automation:** MaxSMT problems can be solved entirely by automated solvers without human intervention, apart from the initial problem specification. This aligns perfectly with the goal of automating network security management.

- **Optimization:** The optimization objectives, such as minimizing the number of allocated firewalls or the number of rules within each firewall, can be directly expressed as soft constraints. The weighted nature of these clauses allows VEREFOO to prioritize different optimization criteria.
- **Formal Correctness:** All critical security requirements and network behavioral rules are encoded as hard constraints. By finding a solution that satisfies all hard constraints, VEREFOO formally guarantees that the resulting firewall allocation and configuration are correct by construction, meaning they inherently adhere to the specified security policies. This eliminates the need for a separate, post-deployment formal verification step.

3.3.3 The Challenge of Modeling for MaxSMT

A significant challenge in using MaxSMT effectively lies in the modeling of the problem components. The formal models of the network (SG, AG, NFs) and security requirements (NSRs) must accurately capture all relevant information that influences the correctness of the solution. If critical details are missing or incorrectly represented, the solver's output, even if mathematically consistent, might not reflect the real-world network's behavior or security posture. At the same time, these models must be designed to keep the number and complexity of the generated MaxSMT constraints manageable. An overly complex model with redundant variables or intricate logical formulas can drastically increase the computational time required by the solver, hindering scalability. VEREFOO strives to find a balance between expressiveness (capturing necessary detail) and complexity (ensuring computational efficiency). For instance, by pre-computing maximal flows (as discussed in the previous section), VEREFOO can reduce the number of free variables that the MaxSMT solver needs to determine, thereby limiting the search space.

3.3.4 Summary of MaxSMT Problem Formulation in VEREFOO

The MaxSMT problem in VEREFOO is constructed from a combination of hard and soft constraints, operating on the formal models of the network, network functions, and security requirements:

- **Hard Constraints:** These clauses ensure the fundamental correctness and adherence to non-negotiable rules.
  - **NSR Satisfaction:** For each network security requirement r, a hard constraint is introduced to ensure it is satisfied. This translates to either the isolation requirement formula or the reachability requirement formula, depending on $r.a$. These formulas ensure that the allocated and configured firewalls correctly enforce the policies.
  - **Network Function Behavior:** Hard constraints define how each network function in the AG (including potential firewalls in APs) forwards

and transforms traffic. For example, for NFs that cannot drop flows, a hard constraint like $deny_i(t) = false$ is applied. For firewalls placed in APs, complex hard constraints define their precise forwarding behavior based on whether traffic should be denied or allowed, considering the firewall's default action and specific rules.

- **Allocation Logic:** Constraints related to the forbidden and forced predicates are hard constraints, ensuring that the AG is correctly constructed and that firewalls are allocated as explicitly required by the user. Additionally, a hard constraint ensures that a firewall is allocated in an AP only if at least one rule is configured within it, preventing the allocation of empty, useless firewalls.

- **Soft Constraints:** These clauses guide the solver towards an optimal solution by assigning weights to preferred conditions.
  - **Minimize Allocated Firewalls:** For each Allocation Place ah, a soft clause is introduced stating that it is preferable for $allocated(a_h)$ to be false (i.e., no firewall is allocated). These clauses are assigned a higher weight to prioritize minimizing the total number of firewalls.
  - **Minimize Rules per Firewall:** For each potential rule (placeholder rule p_i) in a firewall at an AP a_h, a soft clause is added stating that it is preferable for $configured(p_i)$ to be false (i.e., the rule is not included in the firewall's configuration). These clauses have a lower weight than those for firewall allocation but are still crucial for optimization.
  - **Minimize Allowed Traffic Flows (for Security-Oriented Specific):** In the security-oriented specific approach, additional soft constraints are introduced to minimize the scope of ALLOW rules (or maximize DENY rules), promoting a more restrictive security posture. These have the lowest priority among optimization goals.
  - **Weight Assignment:** The weights assigned to soft clauses are carefully chosen to reflect the optimization priorities (e.g., minimizing firewalls is more important than minimizing rules, which is more important than minimizing allowed traffic).

The MaxSMT solver takes all these hard and soft constraints as input. Its output includes the values for the allocated predicate (indicating where firewalls are placed) and the configured predicate (specifying which rules are active in each firewall). Crucially, it also determines the specific 5-tuple-based conditions and actions for these configured rules, providing a complete and deployable firewall configuration.

Chapter 4

Thesis Objective

As detailed in Chapter 3, VEREFOO's core functionality relies on encoding network verification and optimization problems as a partial weighted MaxSMT problem. This approach guarantees the formal correctness of the solution, but it is inherently limited by the computational complexity of the underlying problem. The MaxSMT problem is classified as NP-complete, which means that in the worst-case scenario, the time required for a solver to find a solution grows exponentially with the size of the network and the number of security requirements. In cybersecurity scenarios may not be acceptable to wait too much time, since it is important to react as fast as possible to a cyber-attacks, for this reason REACT VEREFOO [6][7][8] was developed to optimize and speed up the reconfiguration of the firewalls in a network, but also in this case the problem remains a single, indivisible task that cannot effectively leverage modern multi-core or distributed computing resources. To overcome the exponential complexity barrier and truly scale the tool, a more fundamental architectural shift is required. This thesis proposes to tackle the scalability challenge by reframing the problem as a set of smaller, parallelizable sub-problems. The central hypothesis is that by intelligently decomposing a large network into smaller, more manageable subnetworks, and then verifying these subnetworks in parallel, the total verification time can be drastically reduced. The proposed methodology is structured in three main phases:

1. **Network Clusterization:** the first task is to use an algorithm to divide the overall network graph into a set of subnetworks (or clusters). The clusterization has to be performed carefully such that inter-subnetwork dependencies and shared resources (e.g., traffic streams crossing partition boundaries) are minimized, as these dependencies introduce complexity in subsequent stages. The partitioning algorithm has to be efficient and fast enough such that parallel preparation of the network can be accomplished quickly.

2. **Parallel Configuration:** Once the network has been clusterized, each subnetwork is given as input to an independent instance of the VEREFOO solver. These instances can be executed concurrently on multiple processor cores or on separate machines in a distributed computing environment. Solving a small instance of MaxSMT tailored to its subnetwork, any VEREFOO instance can find a solution much more quickly than a single instance would on the entire network.

3. **Aggregation:** The last and most critical step involves aggregating verification results from all parallel executions of VEREFOO and generating a consistent, overall solution of the entire network. This merging process should handle all potential conflicts or inconsistencies induced by isolated subnetwork analysis, in particular regarding flows that traverse partition borders. The aim is to generate a individual, integrated firewall allocation and configuration that is technically correct for the entire network.

The main contribution of this thesis is the design, implementation, and evaluation of this parallelized approach. The expected outcome is a significant reduction in the verification time of VEREFOO, transforming it from a powerful but computationally intensive tool into a scalable and practical solution for large-scale network security management. The drawback of this parallelized approach is the loss of optimality of the solution, which will likely have more firewalls and rules compared to the optimal solution.

Chapter 5

Network Partitioning Approach

This chapter provides a detailed explanation of the proposed approach to divide a large network into multiple subnetworks or clusters, which can then be verified in parallel by independent instances of VEREFOO. The first step will be the introduction of a particular graph called Flow Graph, which is derived from the Service Graph or the Allocation Graph and the set of traffic flows, this representation highlights the density and distribution of traffic flows. Then will be introduced the Leiden algorithm, a graph community detection algorithm fundamental to obtain different clusters from the original network. This algorithm takes in input a network modelled as a graph and returns for each node the cluster it belongs to. The Leiden algorithm will be used on the Flow Graph, it is supposed to create clusters with a high density of traffic flows and minimize the number of traffic flows that pass through more than one cluster (cutoff flows). At this point is possible to use VEREFOO on each obtained cluster and use multi-threading to parallelize the tasks. A crucial step is the management of the cutoff flows, since these flows belong to more than one cluster there is the need to define how the clusters that have in common a traffic flow handle this situation. For each cluster VEREFOO returns an optimal solution, the final step is to merge the different solutions to make a unified solution.

5.1 Flow Graph

The Flow Graph is an undirected weighted graph that is used as the starting point for the network partitioning process. It is derived from the network's logical representation, either the Service Graph (SG) or the Allocation Graph (AG), and the set of traffic flows. The nodes of the Flow Graph are identical to the nodes in the SG or AG. The crucial difference lies in its edges and their assigned weights. For every traffic flow that passes through a link between two nodes in the original SG or AG, the weight of the corresponding edge in the Flow Graph is increased by one. This simple but effective model allows the Flow Graph to visually and quantitatively represent how traffic is distributed across the network, with thicker, more heavily weighted edges indicating areas of greater traffic density. By focusing on flow density rather than just network topology, the Flow Graph is perfectly suited for identifying natural clusters within the network's traffic patterns. Here follows

an example to clarify how to obtain the Flow Graph from the original network and the set of security requirements. Let's consider a simple Service Graph composed of just 8 nodes shown in Fig. 5.1 and the set of requirements in table 5.1

Table 5.1. Example of Network Security Requirements

IP Src	Port Src	IP Dest	Port Dest	Protocol	Action
10.0.0.*	*	10.0.0.*	*	*	Isolation
10.0.1.*	*	10.0.1.*	*	*	Isolation
10.0.0.2	*	10.0.1.*	*	*	Reachability
10.0.1.1	*	10.0.0.2	*	*	Isolation

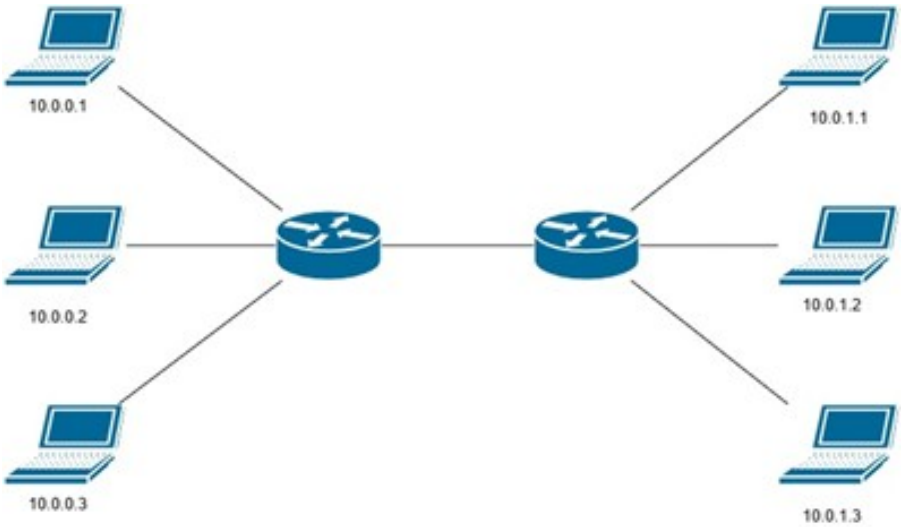

Figure 5.1. Service Graph

From the set of requirements the traffic flows are generated, each traffic flow is represented with an arrow in Fig. 5.2.

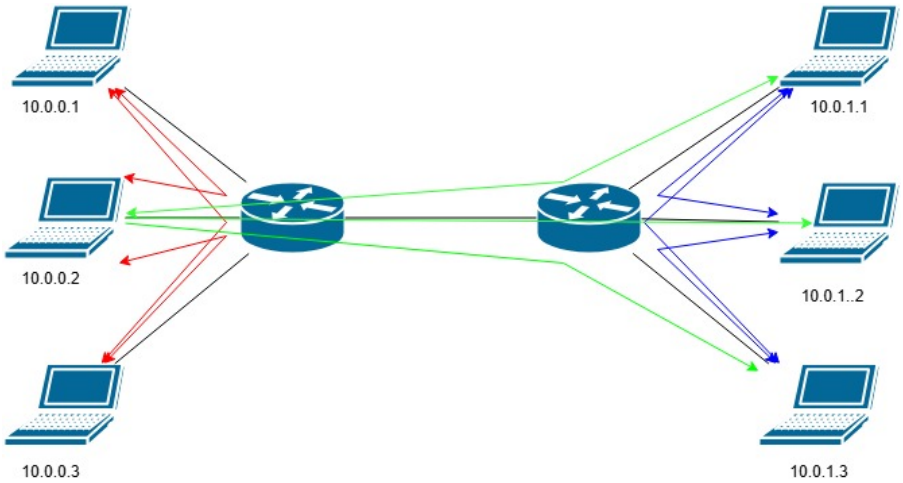

Figure 5.2. Service Graph with flows

At this point is possible to construct the Flow Graph (Fig. 5.3), which will have the same nodes of the Service Graph and each edge will have a weight equal to the number of flows passing through the corresponding edge in the Service Graph.

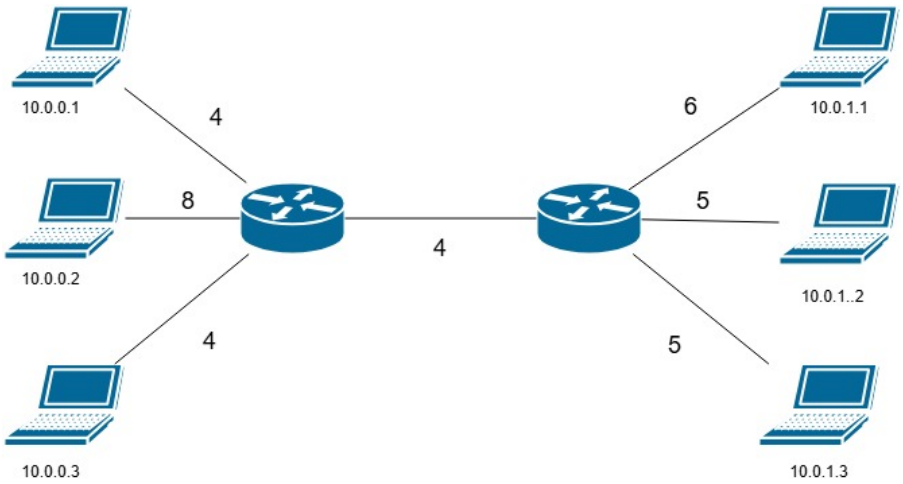

Figure 5.3. Flow Graph

5.2 The Leiden Algorithm

The Leiden algorithm[9] is a state-of-the-art method for detecting communities within a large graph. It is an improvement upon the popular Louvain algorithm and is known for its ability to produce high-quality, well-connected communities, guaranteeing that every community is itself connected. The algorithm has a time complexity roughly of $O(m \log(n))$, where m is the number of edges and n is the number of nodes, making it suitable for partitioning also large networks. The core principle behind community detection is to partition a network's nodes into groups where connections are denser within the groups than between them. The Leiden

algorithm achieves this by iteratively refining an initial, random partitioning. The algorithm's effectiveness is determined by its ability to optimize a quality function, which measures the quality of a given partition. Two common quality functions are modularity and the Constant Potts Model (CPM).

- **Modularity:** This function measures the difference between the number of edges within communities and the expected number of edges in a randomized network with the same number of nodes and edges. A higher modularity score indicates a better community structure. The modularity of a partition P is formally defined as:

$$Q = \frac{1}{2m} \sum_{C \in P} \sum_{i,j \in C} \left[A_{ij} - \frac{k_i k_j}{2m} \right]$$

  where m is the total number of edges in the graph, A_{ij} is the weight of the edge between nodes i and j, and k_i is the sum of the weights of all edges connected to node i. The term A_{ij} accounts for the actual number of edges, while $\frac{k_i k_j}{2m}$ represents the expected number of edges. However, modularity is known to have a "resolution limit," which means it may fail to identify small communities in very large networks.

- **Constant Potts Model (CPM):** The CPM is a more flexible alternative that overcomes the resolution limit. Instead of comparing the graph to a randomized network, it compares the number of edges within a community to a given constant density parameter γ. The CPM value is a sum over all communities of the number of internal edges minus γ times the number of nodes in the community. The CPM for a partition P is defined as:

$$H(P) = \sum_{C \in P} \left[E(C, C) - \gamma \binom{|C|}{2} \right]$$

  where $E(C, C)$ is the total weight of edges within community C, and $|C|$ is the number of nodes in that community. By adjusting the γ parameter, it is possible to control the size and number of the resulting communities. A higher γ value favors smaller, more tightly connected communities, while a lower γ value tends to merge communities into larger ones.

The algorithm's procedure is divided into three main phases that are repeated until no further improvements can be made:

1. **Local Moving of Nodes:** The algorithm starts with an initial partition, often where each node is in its own community. It then iterates through the nodes and moves each one to a neighboring community if that move results in the largest increase in the chosen quality function. This is similar to the first phase of the Louvain algorithm. The key difference in Leiden's approach is that it performs this local moving on a randomly ordered subset of nodes rather than all of them, which makes it more efficient.

2. **Refining the Partition:** Once a locally optimal partition is found, the Leiden algorithm takes a crucial step that distinguishes it from Louvain. It builds a new aggregate graph where each community from the current partition becomes a single, new node. The edges between these new nodes represent the sum of the weights of all edges connecting the original communities. The algorithm then applies the same local moving procedure to this new, smaller, aggregate graph. This process is repeated until a stable state is reached.

3. **Splitting Communities:** This is the most important phase and a key innovation of the Leiden algorithm. After refining the partition, it checks the connectivity of each community. If a community is found to be internally disconnected (meaning it consists of multiple sub-components), the algorithm will automatically split it into its connected components. The quality function used by the algorithm is designed to ensure that splitting a disconnected community will always increase the overall quality score, guaranteeing that the final communities are all connected. This addresses a major flaw in the Louvain algorithm, which could produce disconnected communities.

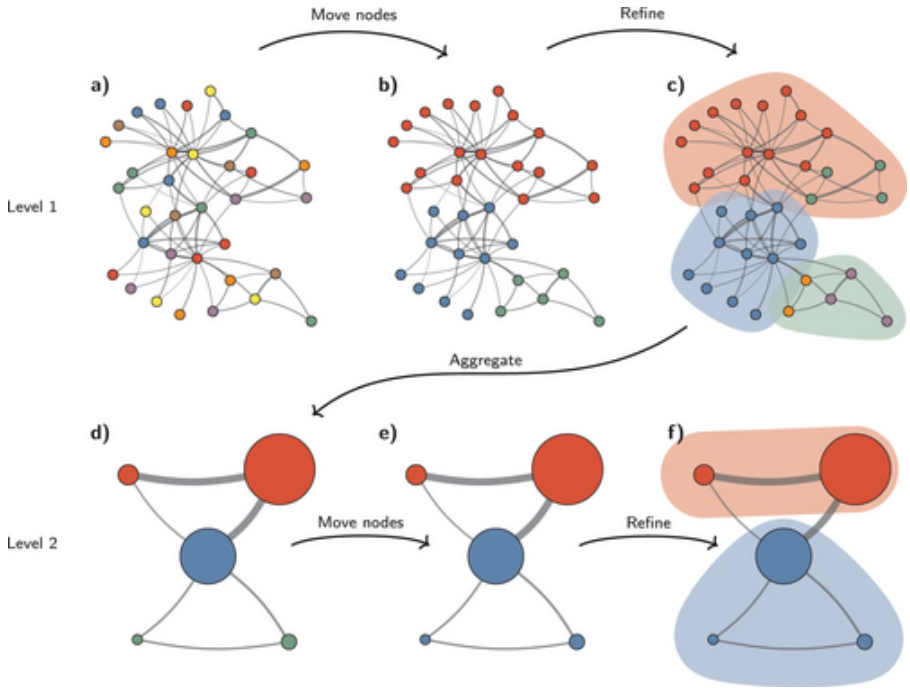

Figure 5.4. Leiden Algorithm

This iterative process ensures that the algorithm converges to a partition where all subsets of all communities are locally optimally assigned, meaning no single node or even a group of nodes can be moved to another community to further improve the quality function. By relying on a fast local move approach and this explicit guarantee of connectivity, the Leiden algorithm provides a powerful and reliable method for partitioning a network into meaningful and well-defined clusters.

5.3 Leiden Application on the Flow Graph

The application of the Leiden algorithm to the Flow Graph is a crucial step in the network partitioning approach. By treating the Flow Graph as the input for community detection, we can identify network regions that are highly cohesive in terms of traffic flows. The algorithm's guarantee of connected communities is paramount in this context; it ensures that a detected cluster of flows truly represents a contiguous segment of the network, preventing nonsensical groupings of disconnected nodes. This effectively transforms a complex, monolithic verification problem into a set of smaller, more manageable sub-problems. The resulting partitions, or subnetworks, are gropus of nodes and traffic flows that can be analyzed independently by separate VEREFOO instances, greatly reducing the computational complexity and enabling a parallelization strategy. In this context, the concept of cutoff flows is particularly important. These are the traffic flows that traverse more than one cluster in the partitioned network. The presence of such flows can introduce suboptimalities into the final solution, specifically by increasing the total number of firewalls and the number of rules required on each firewall. This is because a single cutoff flow must be managed by every cluster it passes through, leading to redundant policy implementations. Therefore, the application of the Leiden algorithm to the Flow Graph serves a critical purpose: by minimizing the number of cutoff flows, it directly contributes to a more optimal and efficient final solution.

5.3.1 Network Partitioning: ideal case

In the ideal case cutoff flows are not present, this means that the Leiden Algorithm can create clusters with no inter-cluster dependencies, leading to an optimal final solution. To give an example, consider the Network Security Requirements in table 5.2, the Service Graph with traffic flows in Fig. 5.5 and the corresponding Flow Graph in Fig. 5.6. Leiden will produce 2 clusters with no cutoff flows as shown in Fig. 5.7.

Table 5.2. Example of Network Security Requirements (ideal case)

IP Src	Port Src	IP Dest	Port Dest	Protocol	Action
10.0.0.*	*	10.0.0.*	*	*	Isolation
10.0.1.*	*	10.0.1.*	*	*	Isolation

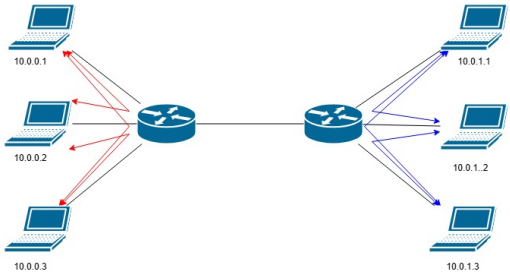

Figure 5.5. Service Graph with flows (ideal case)

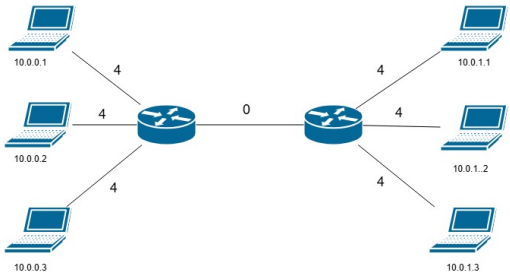

Figure 5.6. Flow Graph (ideal case)

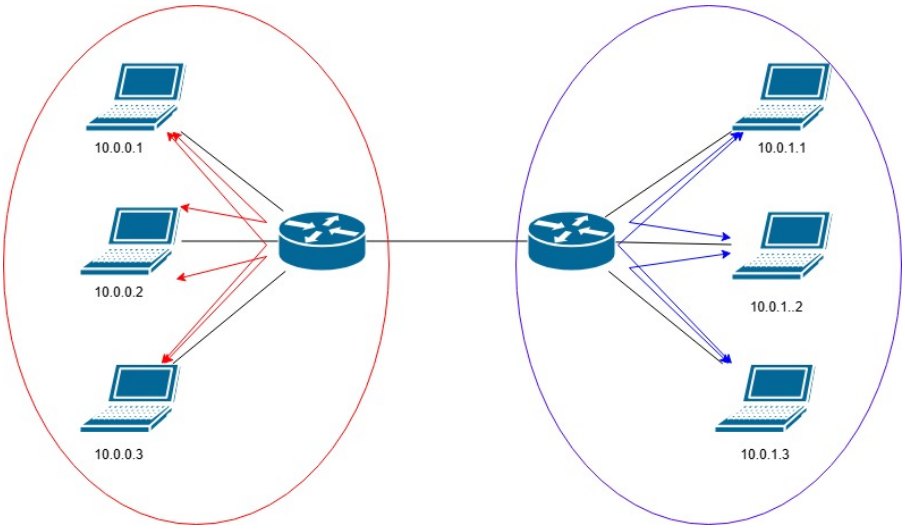

Figure 5.7. Cluster 1 on the left, cluster 2 on the right (ideal case)

Each traffic flow, along with its corresponding Network Security Requirement (NSR), is assigned to a specific cluster. For instance, in Fig. 5.7, the red traffic flows originating from the first NSR in Table 5.1 are assigned to Cluster 1. Similarly, the blue traffic flows from the second NSR are assigned to Cluster 2. This partitioning allows two independent instances of VEREFOO to be applied in parallel, each processing its assigned cluster to find an optimal solution, each VEREFOO instance takes as input the set of nodes within its cluster the associated NSRs and traffic flows. As this represents an ideal case, the combination of these solutions results in a globally optimal outcome.

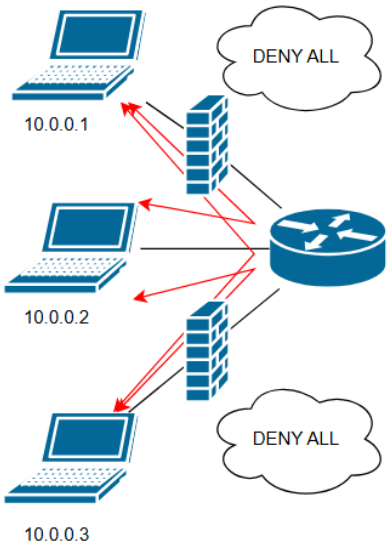

Figure 5.8. Solution for cluster 1

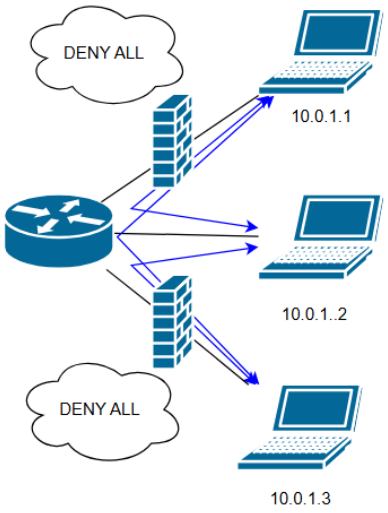

Figure 5.9. Solution for cluster 2

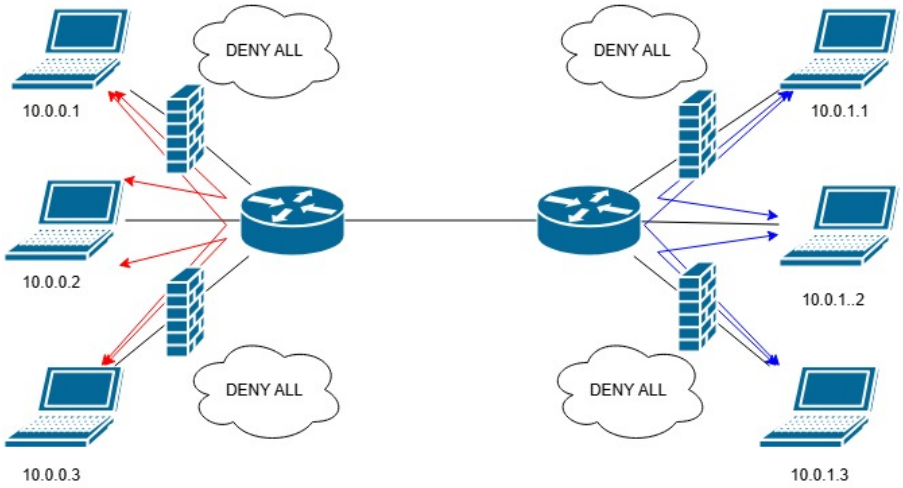

Figure 5.10. Final solution (ideal case)

5.3.2 Network Partitioning: not ideal case

In real-world scenarios, a network is rarely so perfectly divisible that it has no cutoff flows. For this reason, the proposed approach must manage situations where cutoff flows exist and define how clusters handle traffic flows they share with other clusters. As in the ideal case, the Leiden algorithm is applied to the Flow Graph to identify the clusters. Traffic flows that are not cutoff flows are simply assigned to the single cluster they belong to, along with their corresponding Network Security Requirements (NSRs). However, if a flow is identified as a cutoff flow, the cluster is

expanded to include the adjacent allocation places it passes through. This means that an allocation place located between two clusters will belong to both clusters. Consequently, both clusters may attempt to place a firewall in that node, which necessitates a specific merging strategy to resolve potential conflicts, a topic that will be addressed in a later section. Cutoff flows are handled differently depending on the Network Security Requirement (NSR) action they descend from:

- **Isolation cutoff flows:** in case the action of the NSR is *DENY*, a possible solution is to assign the NSR to all the clusters where the associated cutoff flows pass through, This forces each VEREFOO instance to block the same flow, which leads to redundant firewall policies and a sub-optimal solution. Consider the set of NSRs in Table 5.3 and the Service Graph (SG) in Fig. 5.1. The Leiden algorithm divides this network into two clusters, and the third NSR creates a cutoff flow that spans both clusters, as shown in Fig. 5.11.

Table 5.3. Example of NSRs for isolation cutoff flows

IP Src	Port Src	IP Dest	Port Dest	Protocol	Action
10.0.0.*	*	10.0.0.*	*	*	Isolation
10.0.1.*	*	10.0.1.*	*	*	Reachability
10.0.1.1	*	10.0.0.2	*	*	Isolation

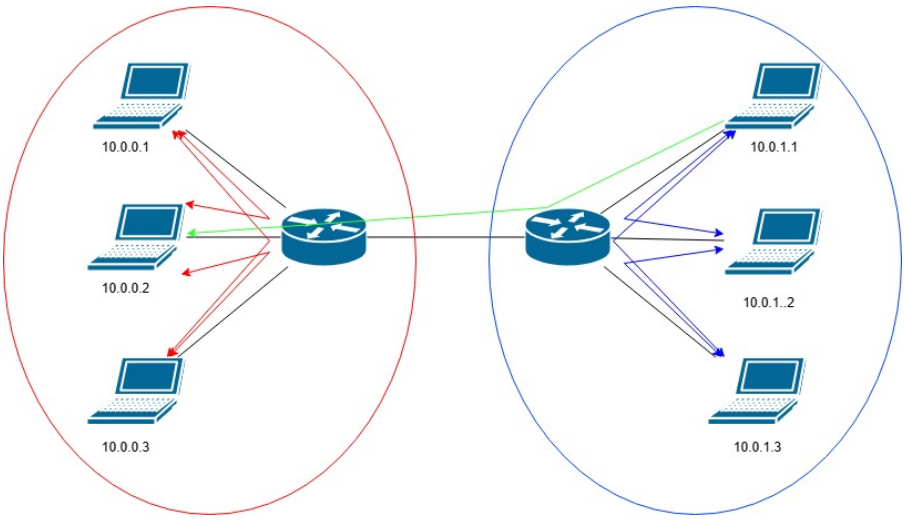

Figure 5.11. Isolation cutoff flow

The SG is then converted into an Allocation Graph (AG) by VEREFOO. Assuming each edge of the SG contains a potential firewall location (allocation place), one such location will be between the two clusters. Since the cutoff flow passes through this location, both clusters must be expanded to include it, as shown in Fig. 5.12, where allocation places are represented by white circles.

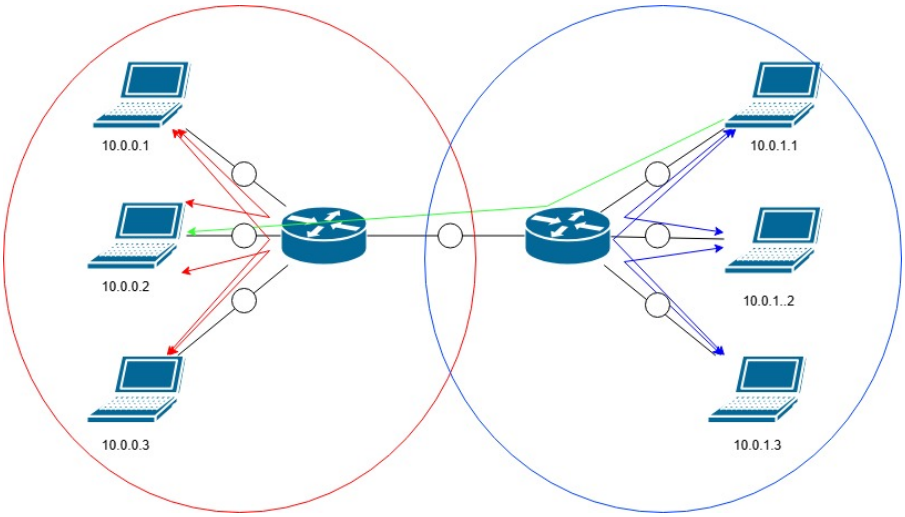

Figure 5.12. Extended clusters

Each cluster will now consider the cutoff flow, but it can only act within its own set of nodes and allocation places. This means the flow is conceptually "cut" at the cluster's border, as seen from the perspective of each cluster (Figs. 5.13 and 5.14). Each VEREFOO instance can only place firewalls within its assigned allocation places.

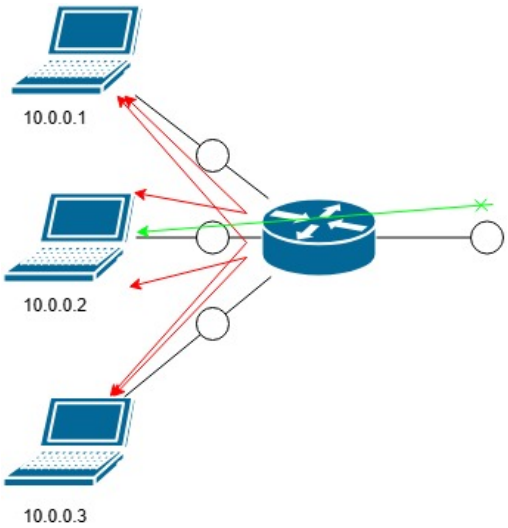

Figure 5.13. Cluster 1 point of view

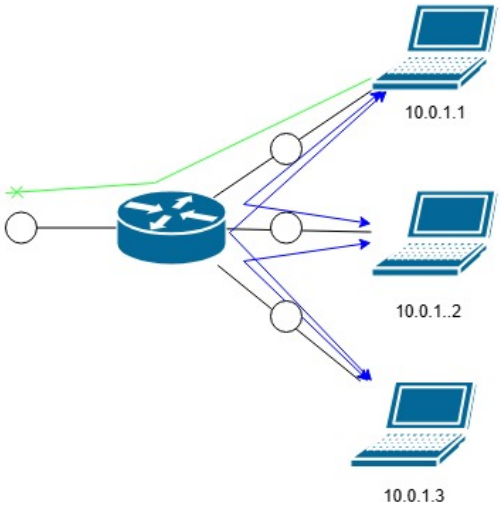

Figure 5.14. Cluster 2 point of view

Each cluster with its own nodes, traffic flows and NSRs is given in input to a different instance of VEREFOO, so each cluster is solved independently as shown in Fig. 5.15 and 5.16.

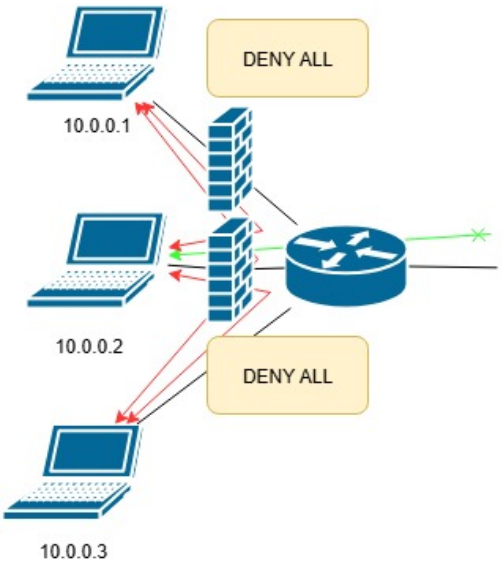

Figure 5.15. Cluster 1 Solution

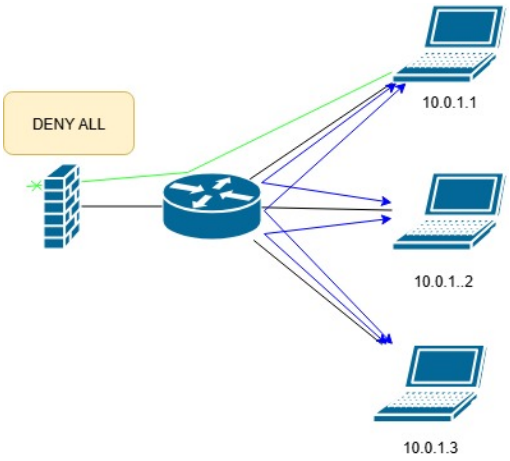

Figure 5.16. Cluster 2 Solution

The original version of VEREFOO, which handles the entire network at once, would solve this optimally by placing only two firewalls as shown in Fig. 5.17. However, this distributed approach solves each cluster independently. Because each VEREFOO instance lacks a complete view of the network, it can lead to a sub-optimal solution where firewalls are redundantly placed, as shown in Fig. 5.18.

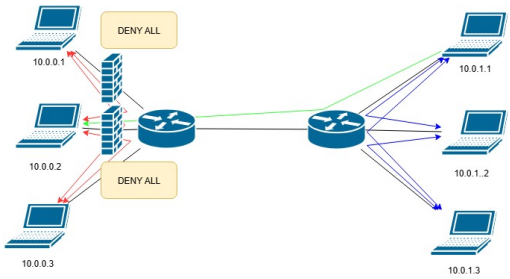

Figure 5.17. Optimal solution

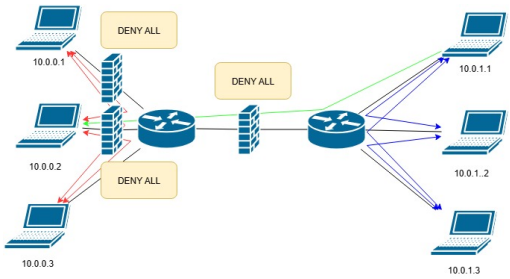

Figure 5.18. Not optimal solution

An alternative solution is to assign the isolation cutoff flow to only one cluster. This is a valid approach because an isolation requirement is satisfied if the traffic is blocked at just a single point along its path. In this case, only one cluster will be responsible for managing and blocking that specific traffic flow.

To decide which cluster receives the assignment, the traffic flow is given to the cluster that has the fewest traffic flows to manage among all the clusters it traverses. This strategy aims to balance the workload across clusters, improving the overall scalability of the solution.

Let's refer again to Fig. 5.12. Both clusters initially have 6 traffic flows to handle (excluding the cutoff flow). Because the clusters are balanced, it is equivalent to assign the cutoff flow to either the first or the second cluster. For this example, let's assign the cutoff flow to the first cluster as shown in Fig. 5.19 and Fig. 5.20.

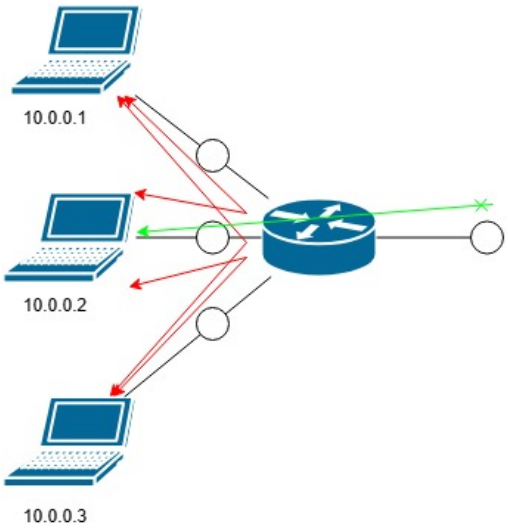

Figure 5.19. Cluster 1 point of view

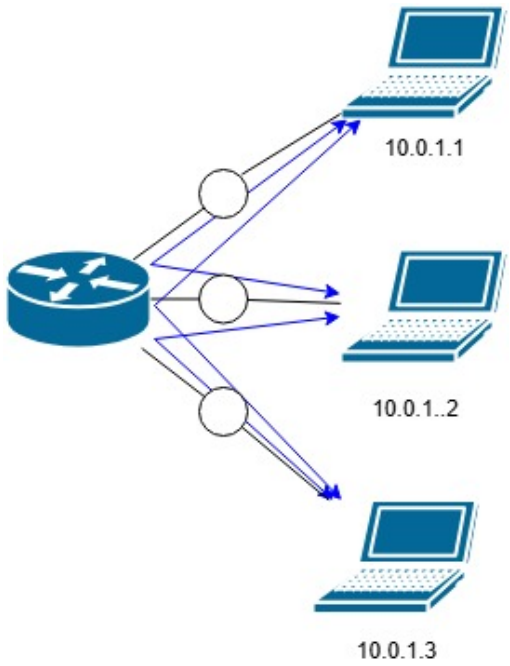

Figure 5.20. Cluster 2 point of view

With this new assignment, the solution for Cluster 1 remains the same as before (Fig. 5.15). However, the second cluster now has no isolation traffic flows to manage and therefore does not need to allocate any firewalls. In this specific case, the final solution will be the optimal one, represented in Fig. 5.17.

This second approach for handling isolation cutoff flows offers two main advantages over the previous method. First, it is more scalable as it aims to balance the number of traffic flows per cluster, preventing situations where one cluster becomes a bottleneck. Second, it is more optimal because assigning the cutoff flow to only one cluster avoids the implementation of redundant policies.

However, it is important to note that even this improved approach does not guarantee a globally optimal solution in every scenario. Since it is not possible to know in advance which cluster will produce the most optimal result after receiving the cutoff flow, the final solution may still have a certain level of sub-optimality. The benefit comes from using a traffic-balancing heuristic to increase the likelihood of a near-optimal outcome, this is why this second solution is preferred and will be used in the final approach.

- **Reachability cutoff flows:** When an NSR's action is *ALLOW*, the NSR is assigned to all clusters that its corresponding cutoff flows pass through. Consider the Service Graph in Fig. 5.1 and the NSRs in Table 5.4. The Leiden algorithm will again divide the network into two clusters. In this case, the third NSR generates three separate cutoff flows, as shown in Fig. 5.21.

Table 5.4. Example of NSRs for reachability cutoff flows

IP Src	Port Src	IP Dest	Port Dest	Protocol	Action
10.0.0.*	*	10.0.0.*	*	*	Isolation
10.0.1.*	*	10.0.1.*	*	*	Isolation
10.0.0.2	*	10.0.1.*	*	*	Reachability

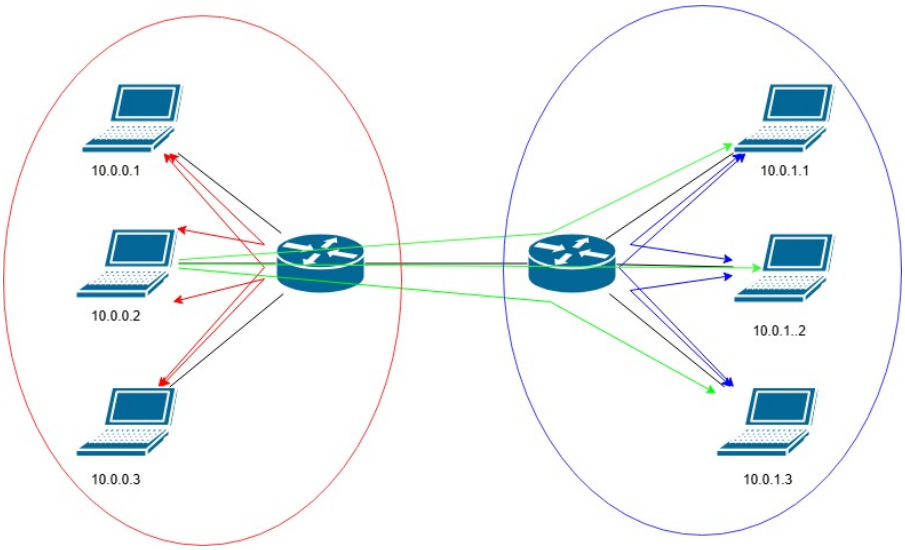

Figure 5.21. Reachability cutoff flows

Just as before, VEREFOO generates the Allocation Graph from the Service Graph, and the clusters are expanded to include the central allocation place because it is traversed by the cutoff flows (Fig. 5.22). For this example, let's assume that the allocation place between node 10.0.0.3 and the forwarder is forbidden for some reason.

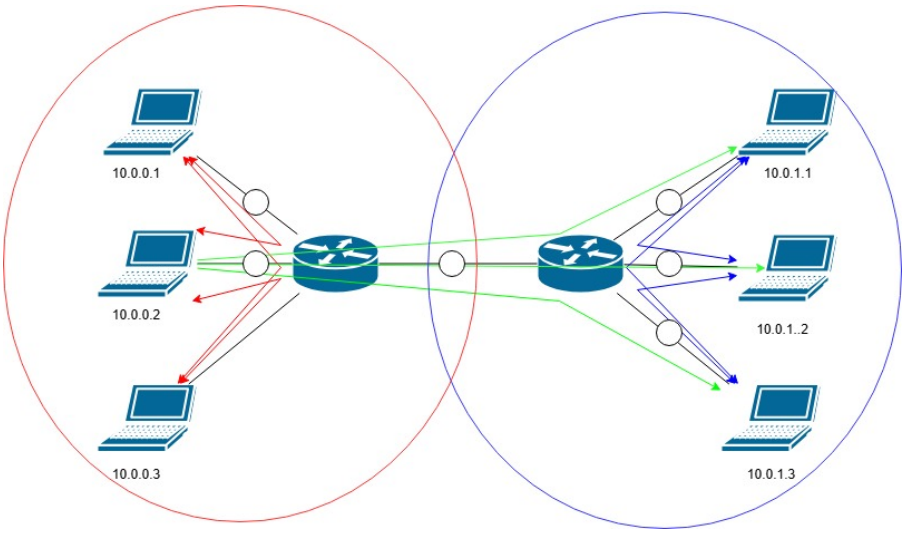

Figure 5.22. Extended clusters

The core challenge with reachability is that, according to the formula, an NSR is satisfied if at least one of its corresponding traffic flows is allowed to pass. Each VEREFOO instance, working in its own cluster, will try to satisfy this condition. However, this can lead to conflicts: one cluster might allow a flow that is then blocked by the second cluster, and vice-versa. As a result, after merging the two solutions, the overall reachability requirement may not be met, as illustrated in Fig. 5.23. In this example, Cluster 1 allows the flow from '10.0.0.2' to '10.0.1.1', but Cluster 2 blocks it. At the same time, Cluster 2 allows the flow from '10.0.0.2' to '10.0.1.2', but Cluster 1 blocks it.

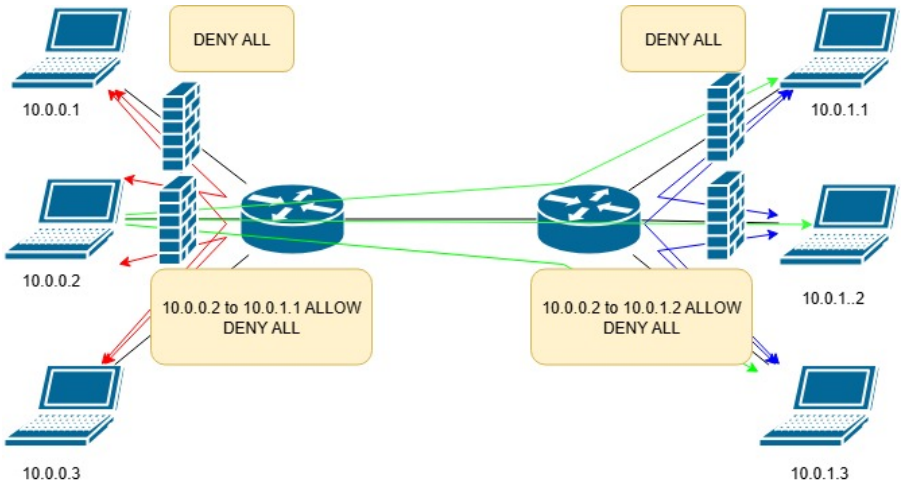

Figure 5.23. Reachability NSR not satisfied

To solve this issue, we can change the NSR's action from "Reachability" to "Complete Reachability". As discussed in Chapter 3, Complete Reachability is a stricter form of reachability where every traffic flow generated by the NSR must be allowed.

This approach guarantees that the final merged solution will satisfy the requirement. However, it can lead to a sub-optimal solution by allowing more traffic flows than are strictly necessary, as seen in Fig. 5.24.

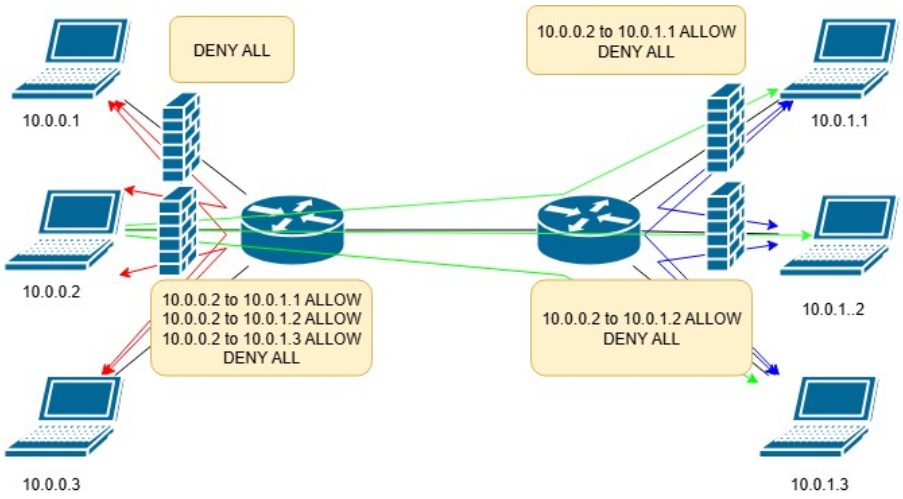

Figure 5.24. Solution with Complete Reachability

5.4 Merge the results

If an allocation place is located between two clusters, it will belong to both of them. This can lead to conflicts if both clusters independently decide to place a firewall in the same location. Consequently, a merge algorithm is necessary to unify the solutions from each cluster into a single, correct final solution. When two clusters both place a firewall in the same shared allocation place, three scenarios are possible:

- **Both firewalls are in denylist:** In this configuration, firewalls are designed to permit all traffic by default and only block specific traffic with explicit rules. As we have established, each isolation cutoff flow is assigned to only one cluster. Each cluster's instance of VEREFOO may then decide to block its assigned isolation cutoff flow using an internal firewall or by placing a firewall at an allocation place that is shared with another cluster. Consequently, the firewalls placed at these shared locations will likely not have identical rule sets. Therefore, the merge operation must combine the rules from both firewalls to create a single, comprehensive rule set that satisfies the requirements of both clusters.

  As an example, let's use the Network Security Requirements (NSRs) from Table 5.5 and the Allocation Graph in Fig. 5.25 after it has been partitioned into two clusters by the Leiden algorithm.

Table 5.5. Example of NSRs for merge of two FWs in denylist

IP Src	Port Src	IP Dest	Port Dest	Protocol	Action
10.0.0.*	*	10.0.0.*	*	*	Reachability
10.0.1.*	*	10.0.1.*	*	*	Reachability
10.0.0.2	*	10.0.1.1	*	*	Isolation
10.0.0.2	*	10.0.1.2	*	*	Isolation

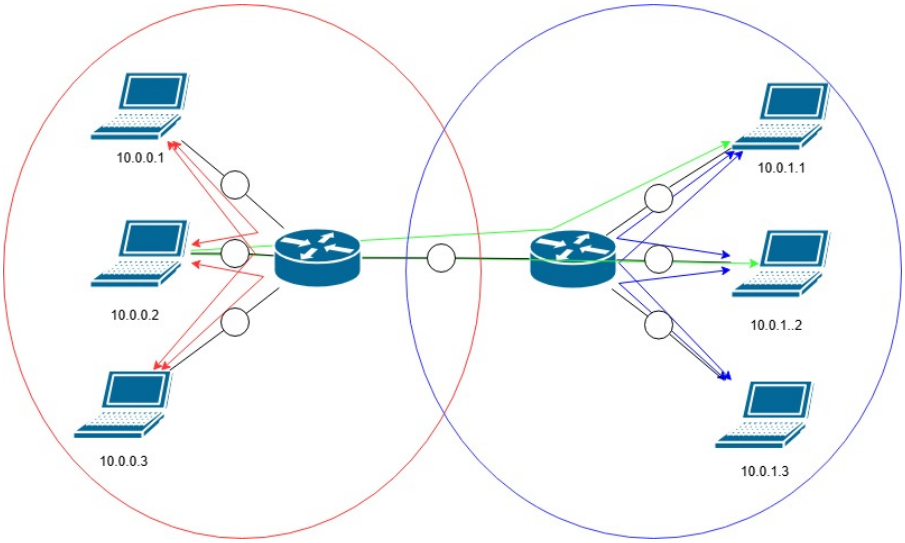

Figure 5.25. Allocation Graph with traffic flows

Excluding the cutoff flows, both clusters have 6 traffic flows to manage. To balance the workload across the clusters, one isolation cutoff flow is assigned to the first cluster and the other is assigned to the second cluster. Each cluster then independently places its firewalls to satisfy its assigned security requirements. As shown in Fig. 5.26 and Fig. 5.27, both clusters place a firewall at the shared allocation place to block their respective cutoff flows.

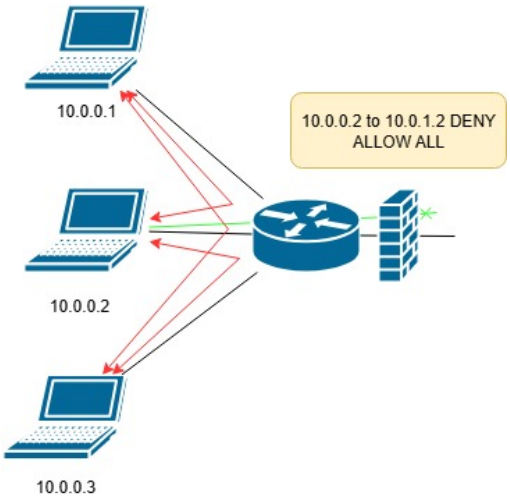

Figure 5.26. cluster 1 point of view

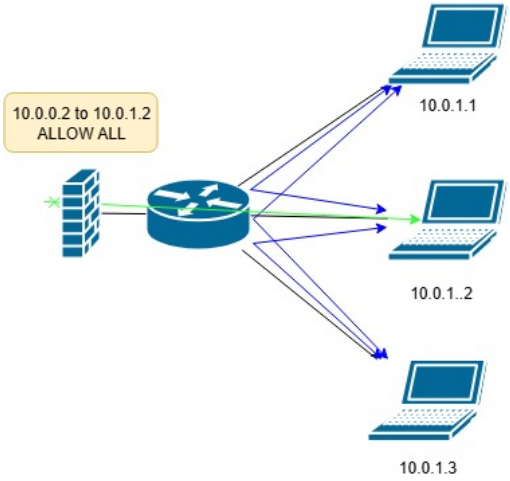

Figure 5.27. cluster 2 point of view

When merging the firewalls from the shared location, their rules are simply combined to create a final, comprehensive rule set that addresses all relevant cutoff flows. The resulting solution is shown in Fig. 5.28.

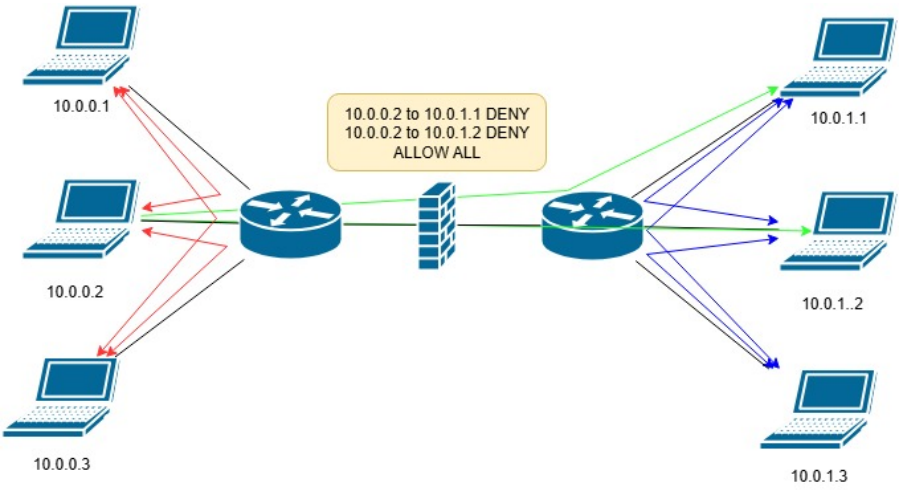

Figure 5.28. Final result

- **Both firewalls use an allowlist policy:** In this configuration, firewalls block all traffic by default and only permit specific traffic that is explicitly defined in their rule sets. Reachability cutoff flows, conversely to isolation cutoff flows, are assigned to both clusters. Therefore, if a cluster decides to place an allowlist firewall in the shared allocation place, it must include specific rules to allow the corresponding reachability cutoff flows. When both clusters independently decide to place an allowlist firewall in the same shared location, their primary goal for that location is identical: to allow the same cutoff flows to pass. This means both clusters will generate the exact same

allow rules to satisfy the shared requirements. However, these firewalls may also have rules not in common, for example if a firewall is allowing some internal reachability traffic flow. Consequently when merging the two solutions, it is sufficient to combine the rules from both firewalls, keeping in mind that there could be duplicate rules which have to be considered only one time.

- **One firewall uses a denylist and the other an allowlist:** This is a particular case because the two firewalls have opposite default behaviors. As discussed, an isolation cutoff flow is associated with only one cluster, while a reachability cutoff flow belongs to both clusters. Since the firewalls have oppisite default actions, one of the two must be converted to the other's type before merging. To dertermine which type of firewall to keep, they are both converted once in allowlist and once denylist and then are merged together. Between two resulting firewalls (one allowlist and one denylist), the one with the fewest rules is chosen for the final solution. To convert a firewall from one policy type to the other (for example, from allowlist to denylist), the following steps are taken:

  1. The firewall's default action is changed. For a conversion from allowlist to denylist, the default action changes from `DENY` to `ALLOW`.

  2. For each traffic flow that crosses the firewall and has an action that is opposite to the *new* default action, a rule is added to manage it.

To make an example, let's consider the firewall in Fig. 5.29, which has an allowlist configuration, and the traffic flows coming form the NSRs in table 5.6. To convert this firewall to a denylist configuration, the default action is changed from `DENY` to `ALLOW`. Then, for each traffic flow that has an action opposite to the new default action (i.e., `ALLOW`), a rule is added to block it. In this case, two rules are added to block the isolation cutoff flows, resulting in the firewall shown in Fig. 5.30.

Table 5.6. Example of NSRs

IP Src	Port Src	IP Dest	Port Dest	Protocol	Action
10.0.0.2	*	10.0.1.1	*	*	Isolation
10.0.0.2	*	10.0.1.2	*	*	Isolation

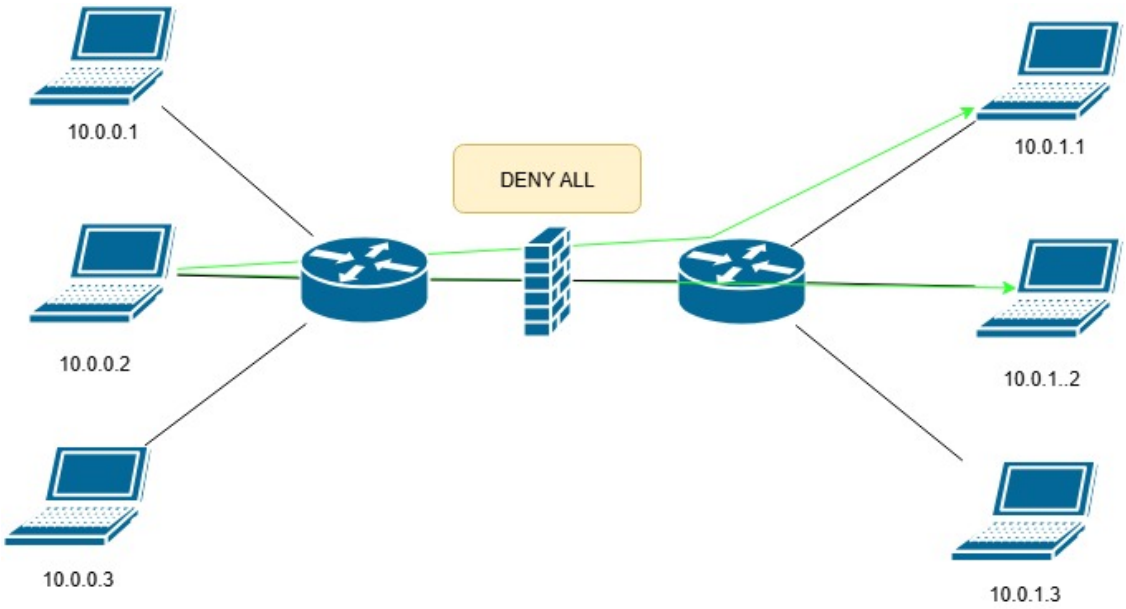

Figure 5.29. Firewall before conversion

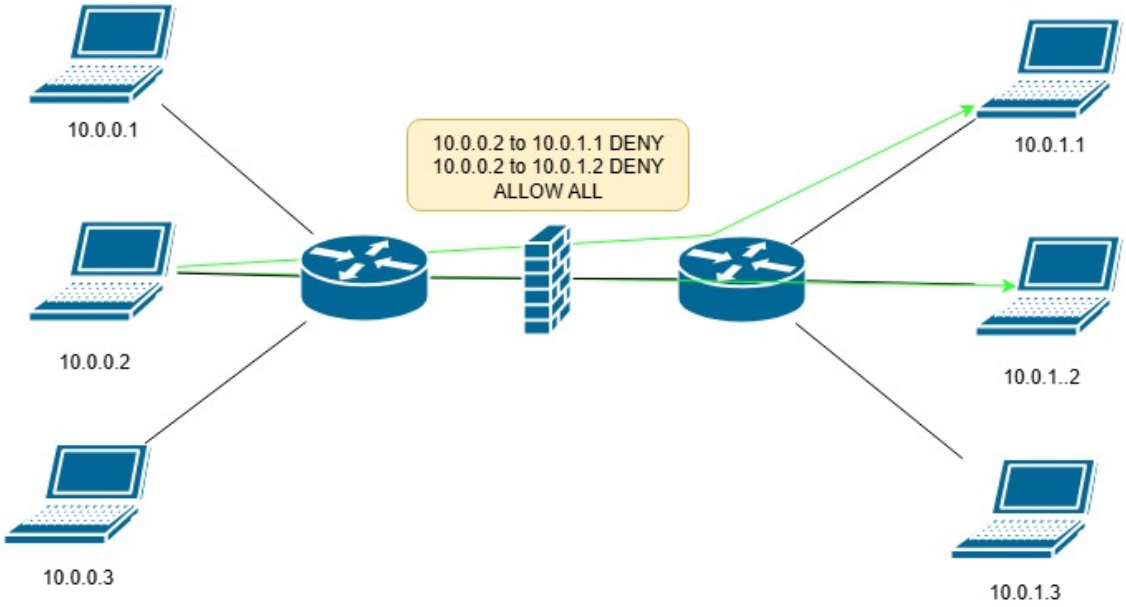

Figure 5.30. Firewall after conversion

Chapter 6

Implementation and Validation

The approach described in chapter 5 has been fully implemented in Java as an extension of the already existing code of VEREFOO. This chapter presents the tests which have been conducted on the implementation of the proposed approach, to show which goals have been achieved and to understand the limitations which should be overcome in the future. The first part of this chapter will focus on the validation of the Leiden algorithm, to understand how the different parameters influence the final clustering and to provide some guidelines for their correct choice. The second part will present the results of the parallelized VEREFOO, comparing its performance and optimality with the standard version.

6.1 Leiden Implementation and Validation

For the Leiden algorithm has been adopted a public available java library [10], this implementation of Leiden requires different parameters in input that can modify the final clusterization. Some of these parameters are left to the default value and cannot be modified:

- **Quality function (CPM or Modularity):** this parameter defines the quality function to use, is set to "CPM" by default which overcomes some limitations of the Modularity
- **Algorithm (Louvian or Leiden):** the implementation allows to choose between the Louvian and the Leiden algorithm, the dafault value is "Leiden" which is an improvement of Louvian
- **Iterations:** defines the number of iterations of the algorithm, by default is set to "10"

Other parameters are not fixed and is network administrator's duty to understand which values best suit for a specific network based on some guidelines that will be presented later in this chapter. These parameters are:

- **Resolution Parameter γ:** is a real number such that $0 < \gamma < \text{inf}$, small values produce less clusters, higher values more clusters
- **MinNodes:** define the minimun number of nodes per cluster
- **Normalization:** can be "None"(N) or "AssociationStrength"(A). Used to normalize the weights of the flow graph, useful for heterogeneous networks in terms of traffic flows

To understand how the previously mentioned parameters influence the final clustering, a series of tests were conducted. The objective was to define a set of guidelines for correctly choosing these parameters to achieve an optimal network clusterization. A "good" clusterization in this context is defined as a partition where the clusters are balanced in terms of the number of nodes, which is quantitatively measured by a low standard deviation of node counts across all clusters. The tests were performed on two types of network created with two different generators, both generators take input the desired number of webclients and the number of policies (NSRs) and produce an allocation graph that differentiates for the topology, in particular:

- The **GEANT generator**, creates star-shaped network topologies composed of a central node with eight-node chains.
- The **VPNConfB generator**, creates more complex, non-star-shaped network topologies.

To evaluate performance across different scales, tests were conducted on networks with the following dimensions:

- 50 Webclients, 150 Policies
- 80 Webclients, 240 Policies
- 100 WebClients, 300 Policies
- 300 Webclients, 900 Policies
- 500 Webclients, 1500 Policies

The Leiden algorithm's parameters were varied within the following ranges during the testing:

- **Resolution Parameter γ:** Varied among 0.1,1.0,2.0.
- **MinNodes:** Varied among 2,5,10.
- **Normalization:** Varied between "N" and "A".

For each network topology, network dimension and combination of parameters have been computed the following statistics:

- Number of clusters
- Standard deviation of the number of nodes per cluster
- Standard deviation of the number of flows per cluster
- Number of cutoff flows

6.1.1 GEANT networks analysis

GEANT networks are star-shaped networks with a central node and chains of 8 nodes as shown in Fig. 6.1, where the allocation places are represented in red and normal nodes in black.

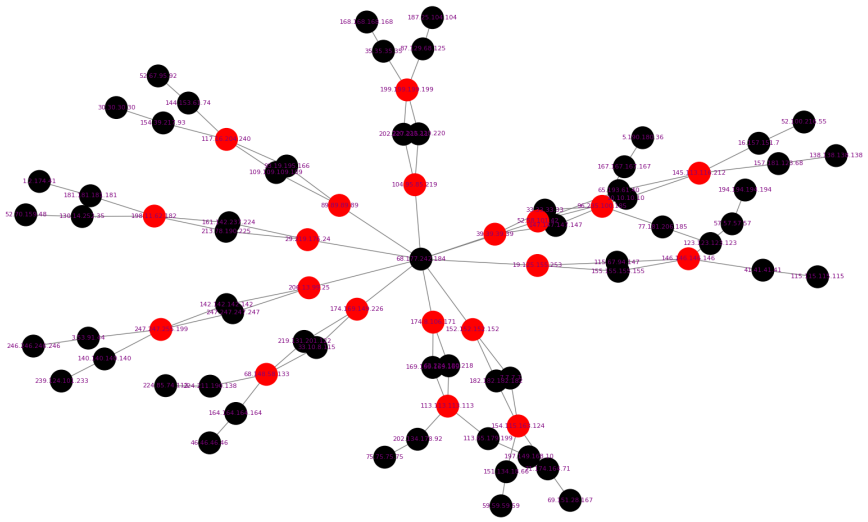

Figure 6.1. Example of GEANT network with 20 Webclients

Due to their structure, these networks are always partitioned into a large central cluster containing the central node, along with several smaller clusters, each ideally corresponding to one of the eight-node chains. Increasing the resolution parameter allows for a reduction in the size of the large central cluster, leading to an increased number of smaller clusters. This generally results in better parallelization performance but decreases the optimality of the final solution. As can be observed in Fig. 6.2 and 6.3, an increase in the resolution parameter corresponds to a higher number of clusters. In this specific case, the standard deviation of the number of nodes per cluster also increases. This is because the MinNodes parameter is set to a low value of 2, which allows for the creation of very small clusters that significantly contribute to the standard deviation.

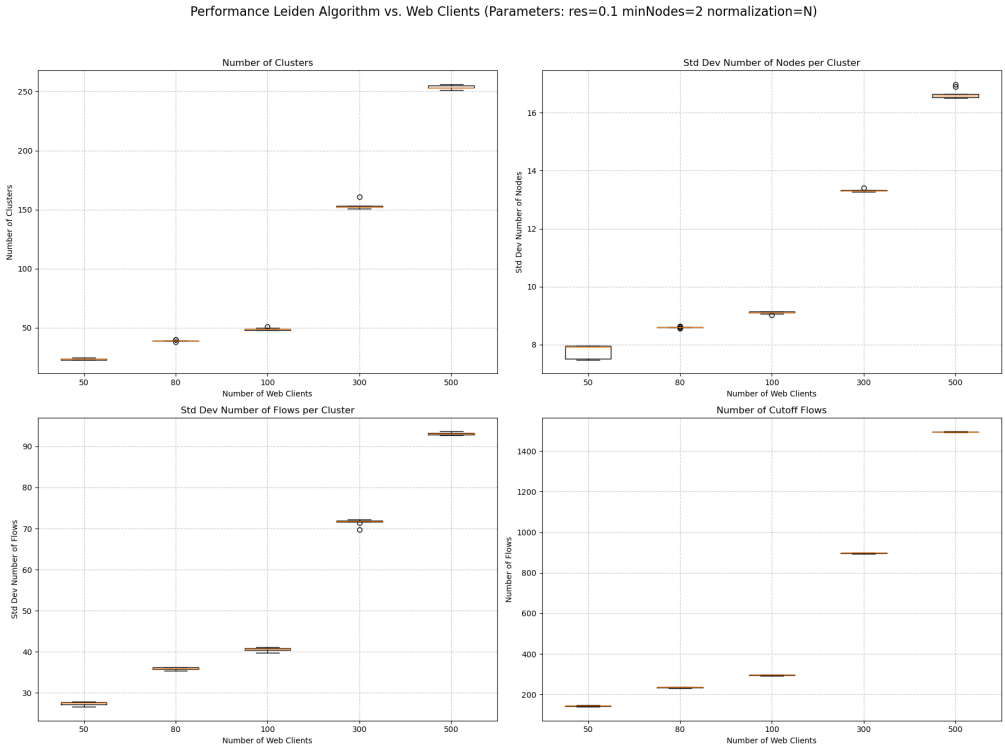

Figure 6.2. GEANT rp=0.1, minNodes=2, norm="N"

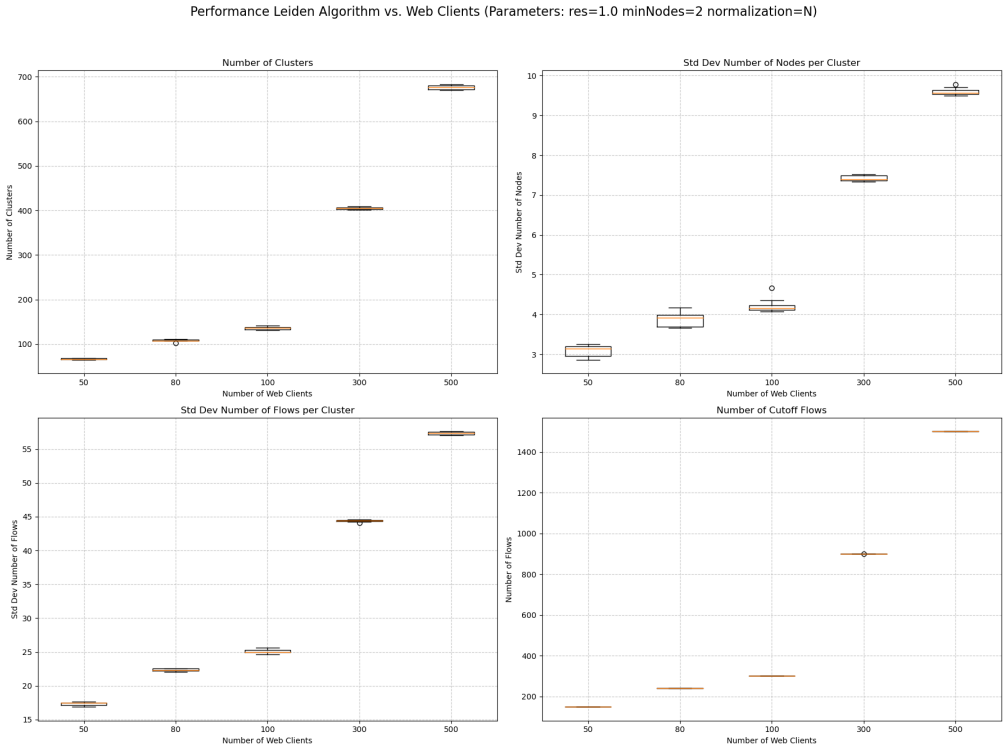

Figure 6.3. GEANT rp=1.0, minNodes=2, norm="N"

To avoid the creation of clusters smaller than 8 nodes it is necessary to set "minNodes" equals to an higher value, keeping unchanged the resolution parameter. This will decrease the number of clusters but will also decrease the standard deviation as shown in Fig. 6.4

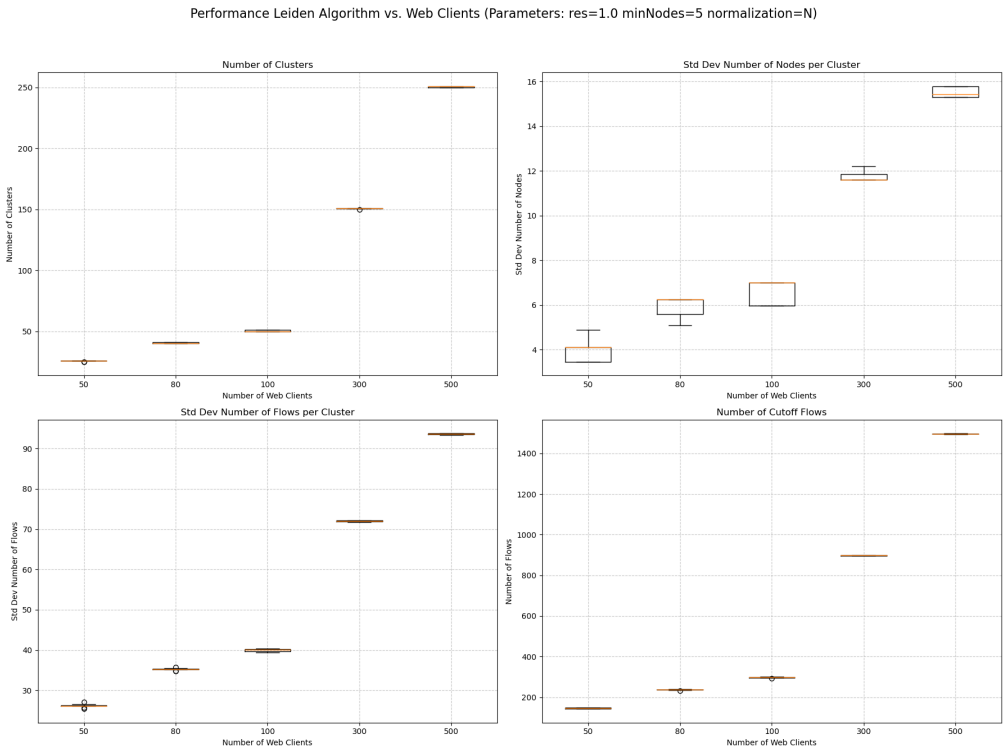

Figure 6.4. GEANT rp=1.0, minNodes=5, norm="N"

If the resolution parameter is increased again as in Fig. 6.5 the number of clusters and the standard deviations do not change across all the network dimensions. This is because the Leiden algorithm, unable to further partition the strongly connected central cluster, attempts to divide the smaller chain clusters. However, since the MinNodes parameter is set to 5, any new clusters smaller than this are reconnected, effectively maintaining the chain-based clusters and preserving a stable partitioning. So for this network topology, after a certain value of the resolution parameter, there may be an intrinsic limit for the Leiden algorithm to further clusterize the network.

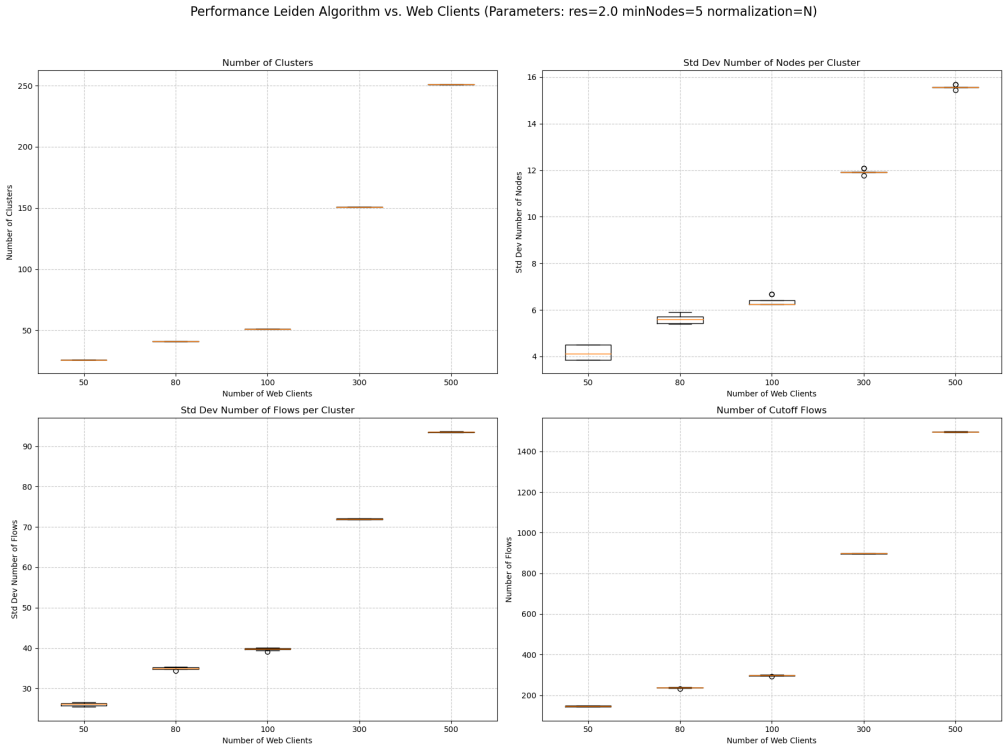

Figure 6.5. GEANT rp=2.0, minNodes=5, norm="N"

As final consideration for this topology, "minNodes" should be set to a value smaller or equal to the the chain size, otherwise Leiden would produce a single cluster containing the entire network, making the problem equals to the standard version of VEREFOO.

6.1.2 GEANT results

Given the previous Leiden analysis on the GEANT networks, it is possible to provide some guidelines to correctly choose the parameters to obtain a good clustering. As said before, the algorithm creates a big cluster containing the central node and many smaller clusters. To ensure that these smaller clusters are balanced with a size equal to the natural chain length, it is sufficient to set the "minNodes" parameter to 8.
The choice of the resolution parameter is based on the primary objective. In the case where the performance is the main objective, a higher resolution parameter, such as 1.0 or 2.0, has to be chosen for the generation of more small and more parallelizable clusters. In the situation where also the optimality is a primary concern, a lower resolution parameter, such as 0.001 or 0.01, has to be employed for the purpose of creating less clusters.
Finally, Normalization is not required for the GEANT network topology. its effect is just reducing the number of clusters, which can be achieved by simply reducing the resolution parameter, making normalization unnecessary for this specific network topology.

6.1.3 VPNConfB network analysis

VPNConfB generator creates more complex networks, an example with 20 web-clients is shown in Fig. 6.6

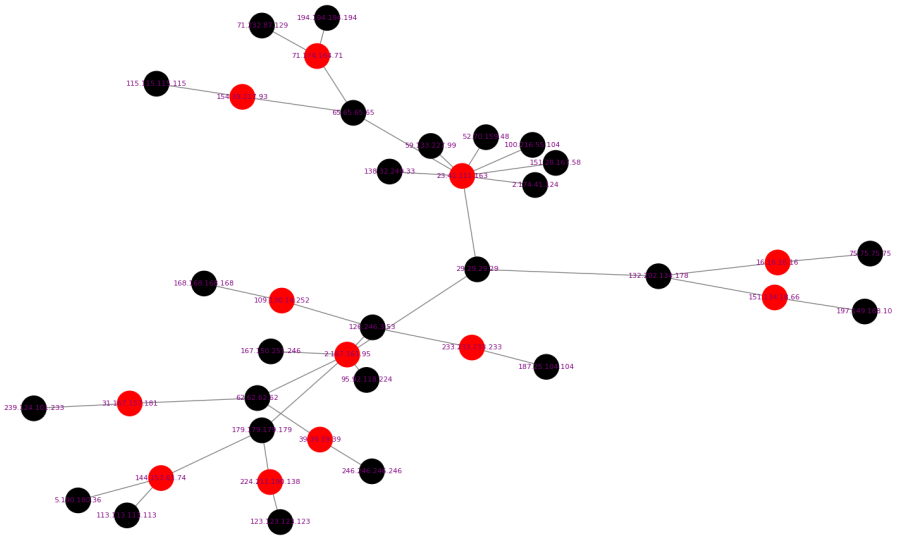

Figure 6.6. Example of VPNConfB network with 20 webclients

These kind of networks are very heterogeneous in terms of traffic flows, which means that there are areas with a high flow density and others with poor density, this results in very unbalanced clusters and a large standard deviation in the number of nodes per cluster as can be seen in 6.7. A counter-intuitive behavior is observed when the resolution parameter increases, as one can see from Fig. 6.7. For larger networks, the number of clusters decreases instead of increasing. This phenomenon is a direct consequence of the network's heterogeneity and its interplay with the MinNodes parameter. The Leiden algorithm, faced with such a heterogeneous distribution of flow, tends to output at a first stage big clusters very interconnected and a vast number of clusters that are composed by a single node. As we have MinNodes set at 2, these one-node clusters are merged together to form larger clusters, thereby causing a reduction in the number of clusters. Furthermore, this merging process exacerbates the imbalance, leading to a significant increase in the standard deviation, indicating a less balanced clustering solution.

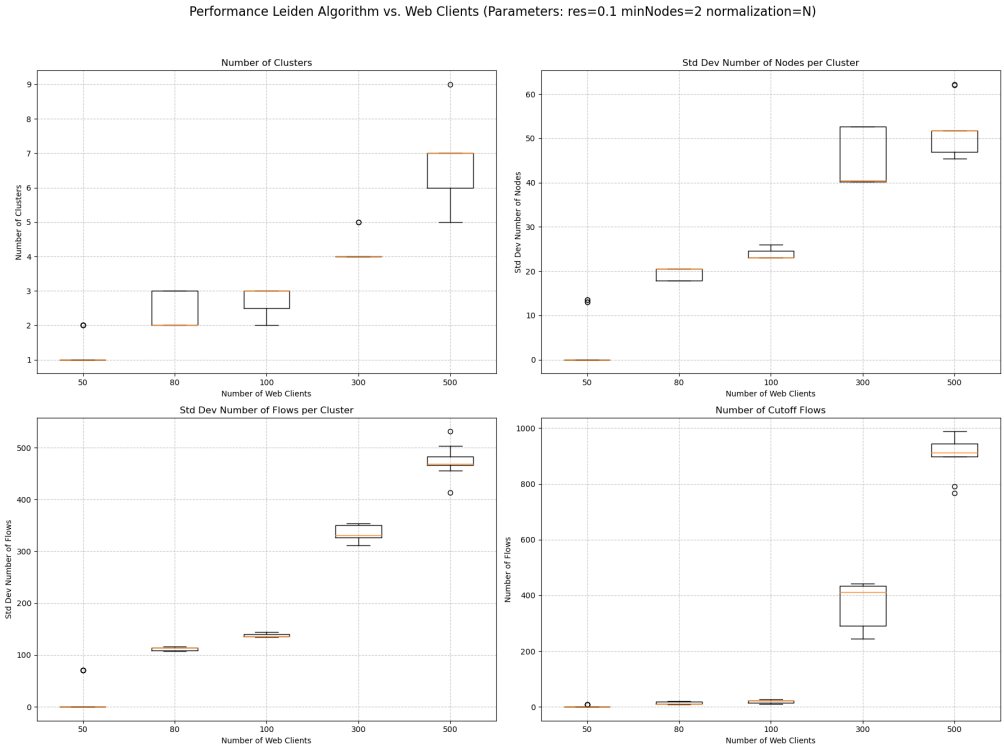

Figure 6.7. VPNConfB rp=0.1, minNodes=2, norm="N"

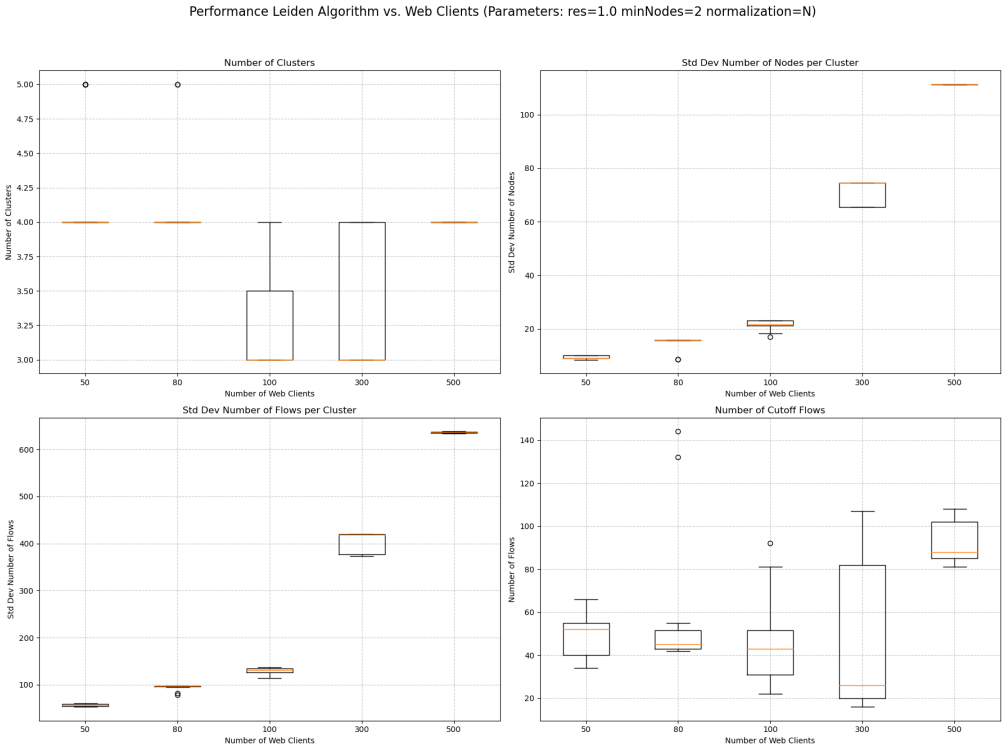

Figure 6.8. VPNConfB rp=1.0, minNodes=2, norm="N"

To achieve more balanced clusters in this network topology, it is essential to utilize the Normalization parameter. This feature normalizes the weights of the flow graph, thereby mitigating the negative effects of the network's high heterogeneity. As a result, shown in Fig. 6.9, the clustering produces a higher number of clusters with a significantly lower standard deviation of the number of nodes per cluster. Further, by increasing the MinNodes parameter, as demonstrated in Fig. 6.10, the total number of clusters decreases, which helps to avoid the creation of very small subnetworks and contributes to a more balanced and robust partitioning.

By increasing again the resolution parameter the clustering does not change (Fig. 6.11), meaning that Leiden is not able to divide the network in more clusters, so as before there is an intrinsic limit in the Leiden algorithm to further clusterize the network.

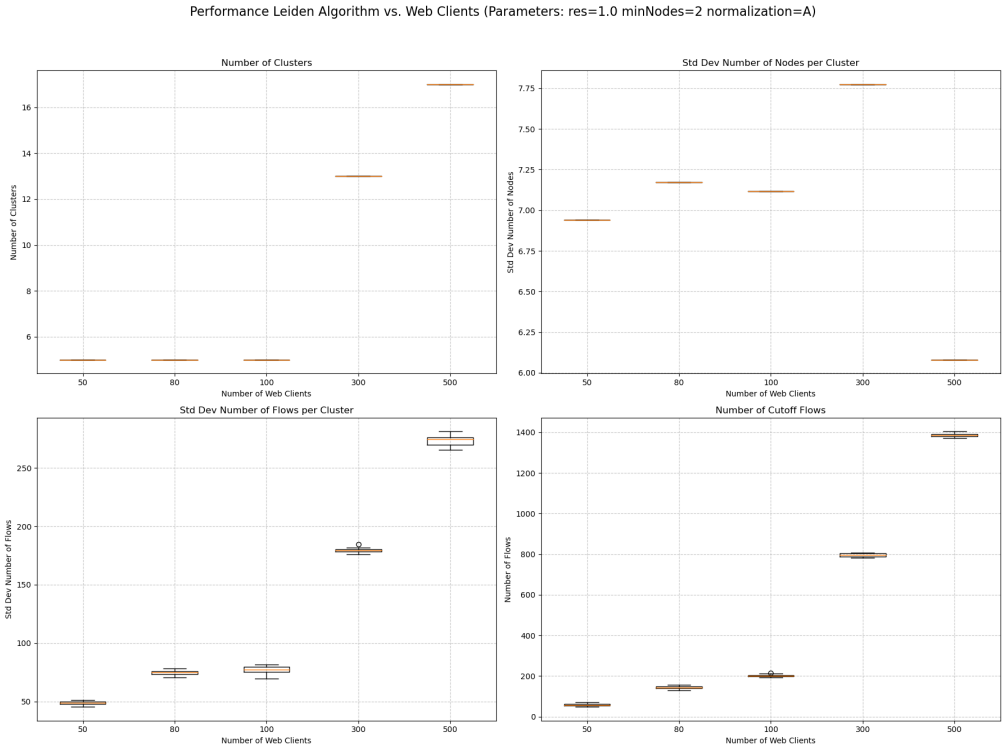

Figure 6.9. VPNConfB rp=1.0, minNodes=2, norm="A"

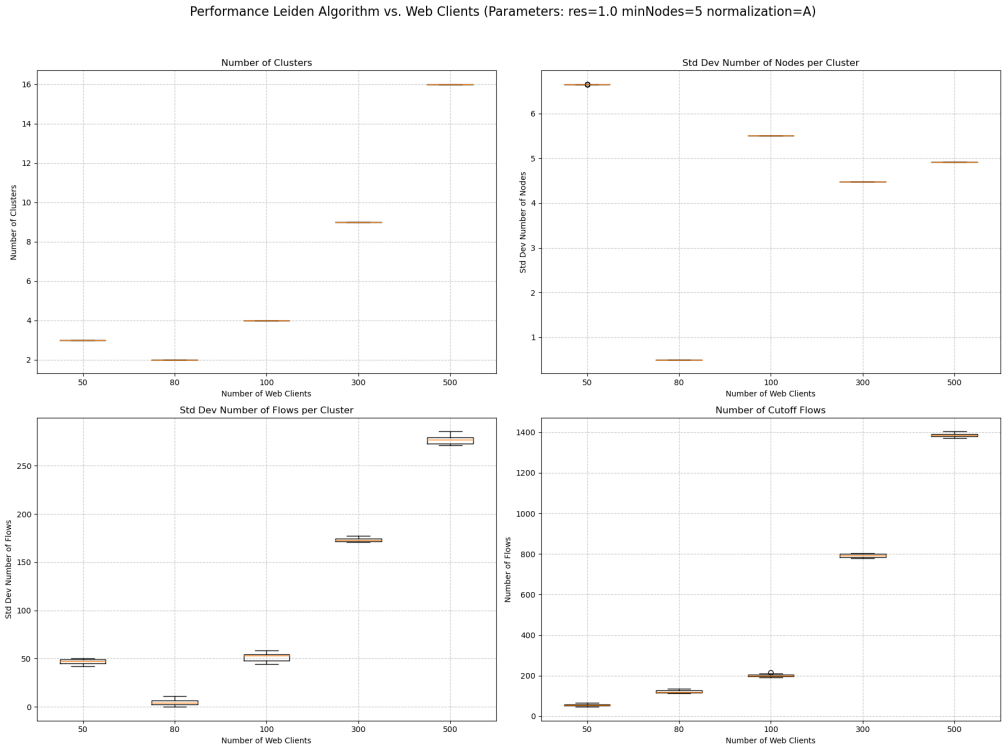

Figure 6.10. VPNConfB rp=1.0, minNodes=5, norm="A"

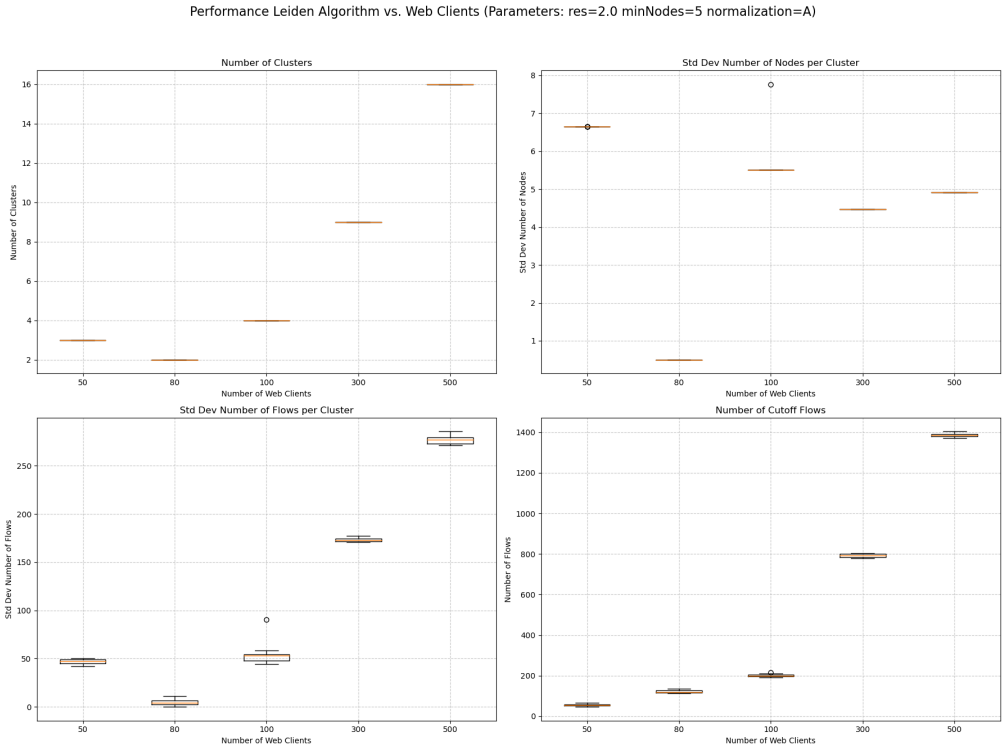

Figure 6.11. VPNConfB rp=2.0, minNodes=5, norm="A"

6.1.4 VPNConfB results

Based on the preceding analysis of VPNConfB networks, we can now establish a set of guidelines for the optimal selection of input parameters for the Leiden algorithm. When the primary objective is to maximize performance, a higher resolution parameter, such as 1.0 or 2.0, is recommended to produce a greater number of parallelizable clusters. Conversely, a lower value should be chosen for a more optimal solution.
To prevent the formation of very small subnetworks and ensure a more balanced partitioning, it is advisable to set the "MinNodes" parameter to a value of at least 5.
Finally, the "Normalization" parameter is crucial for achieving balanced clusters within this network topology, as it effectively counteracts the negative effects of high traffic heterogeneity.

6.2 Performace Validation of Parallel VEREFOO

In this section are analyzed the performance and the optimality of parallel VEREFOO against the standard version. The tests were conducted on the same network topologies used for the validation of the Leiden algorithm, specifically GEANT and VPNConfB networks. The aim of this analysis is to compare the two versions of VEREFOO until reaching the maximum network size that the standard version can handle, subsequently evaluating the performance of the parallel version on larger networks. The tests were performed on a Ubuntu virtual machine with 4 CPU cores and 8 GB of RAM, the processor of the host machine was an Intel i7-13620H.

6.2.1 GEANT networks performace

For all the tests on GEANT networks, the parameters of the Leiden algorithm were set as follows: resolution parameter = 1.0, MinNodes = 8, Normalization = "N". These values were chosen based on the guidelines established in the previous section to ensure a good clustering.
In the first part of the analysis, the performace of the two versions of VEREFOO were compared on networks with dimensions ranging from 10 to 50 webclients, from this value the standard version cannot provide a solution before the timeout threshold of 30 minutes. For each network dimension the number of policies has been computed as $numberWebclients \times 3$. The results are shown in Fig. 6.12 for Atomic Flows and in Fig. 6.13 for Maximal Flows. The parallelized version shows a significant performance improvement, being able to solve in few seconds networks that require several minutes with the standard version.

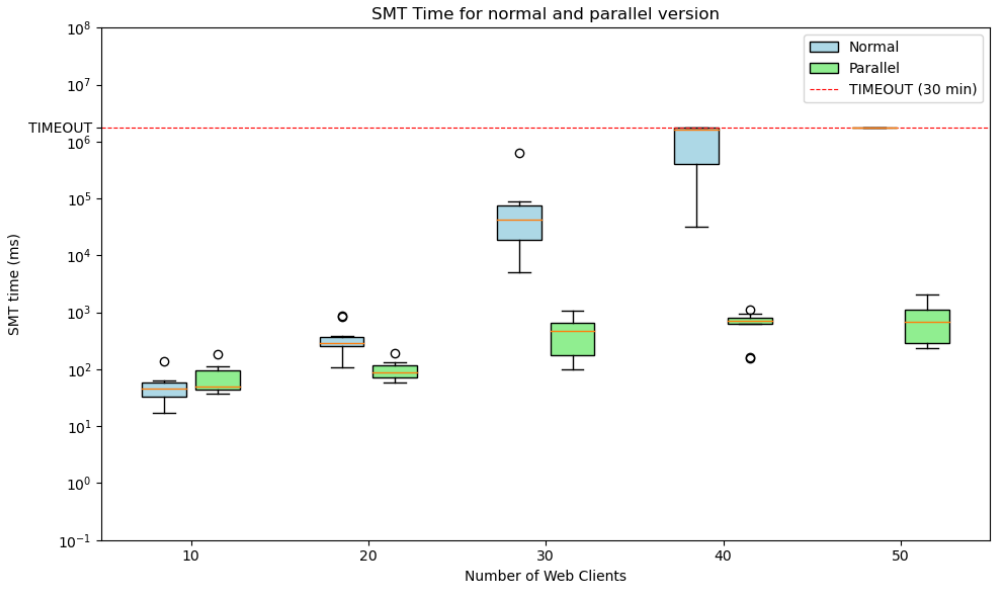

Figure 6.12. SMT time comparison standard vs parallel version with Atomic Flows (GEANT)

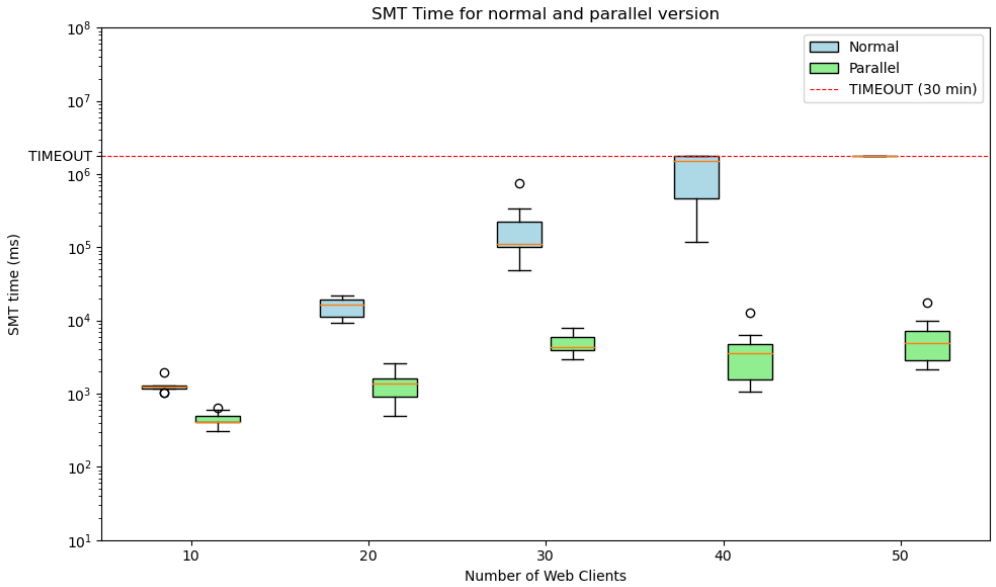

Figure 6.13. SMT time comparison standard vs parallel version with Maximal Flows (GEANT)

However, the parallel version does not provide the optimal solution as discussed in chapter 5. The optimality of the two solutions is compared in Fig. 6.14 and 6.15 for Atomic Flows and in Fig. 6.16 and 6.17 for Maximal Flows, where are shown the number of firewalls and average number of rules used in the two versions.

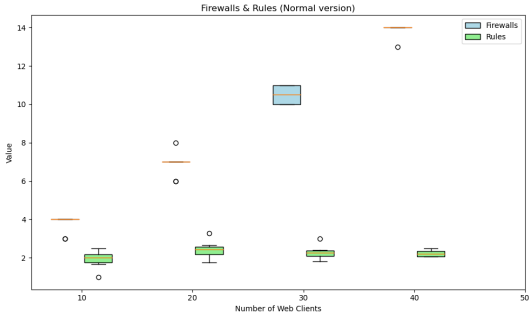

Figure 6.14. firewalls and rules normal version with Atomic Flows (GEANT)

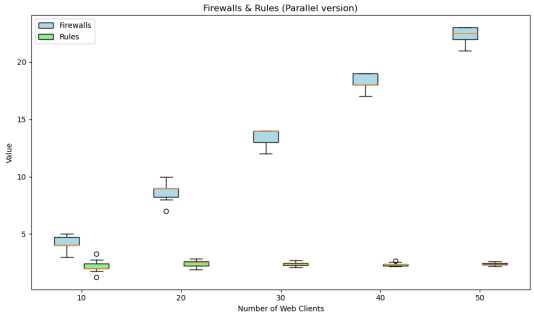

Figure 6.15. Firewalls and rules parallel version with Atomic Flows (GEANT)

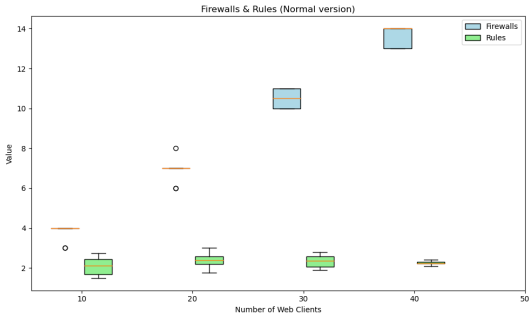

Figure 6.16. Firewalls and rules normal version with Maximal Flows (GEANT)

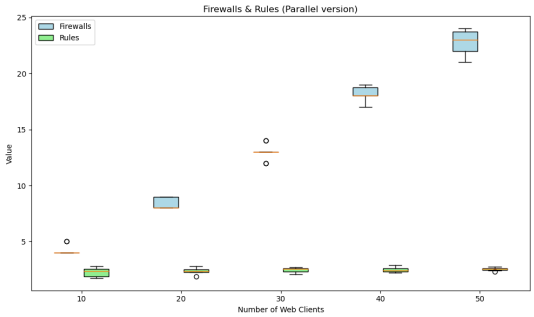

Figure 6.17. Firewalls and rules parallel version with Maximal Flows (GEANT)

The parallel version has been further tested in larger networks, from 100 webclients up to 1000 webclients. It was not possible to test on networks larger than this size because of the complexity in computing the Atomic Flows due to a limited amount of RAM. Again, the number of policies has been computed as $numberWebclients \times 3$. The results are shown in Fig. 6.18 for Atomic Flows and in Fig. 6.19 for Maximal Flows. These results show that the parallel version is able to solve networks up to thousands of nodes in few seconds or minutes for the Atomic Flows, while for Maximal Flows the maximum network size that can be solved in reasonable time is 500 webclients.

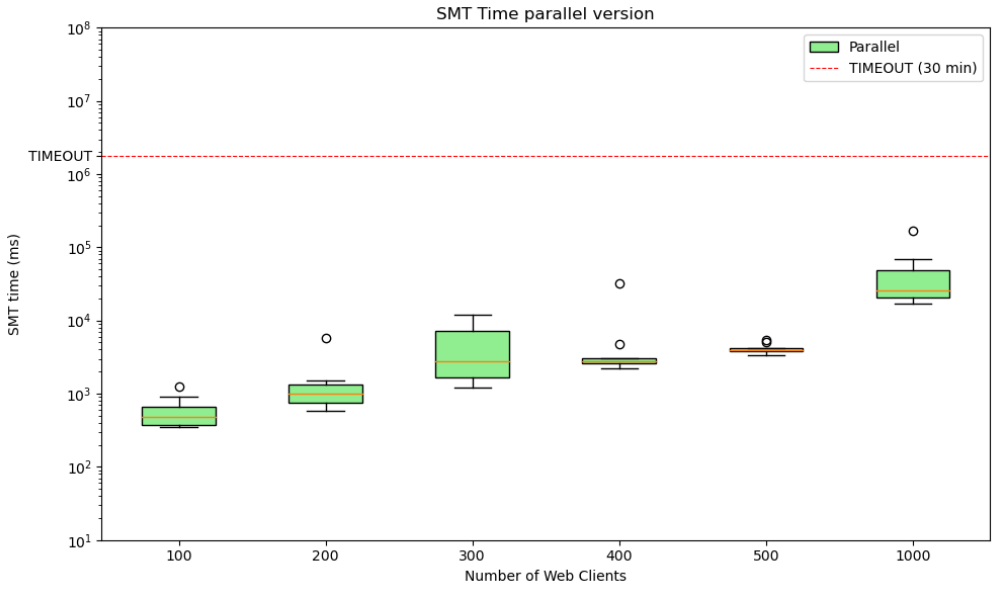

Figure 6.18. SMT time parallel version for large networks with Atomic Flows (GEANT)

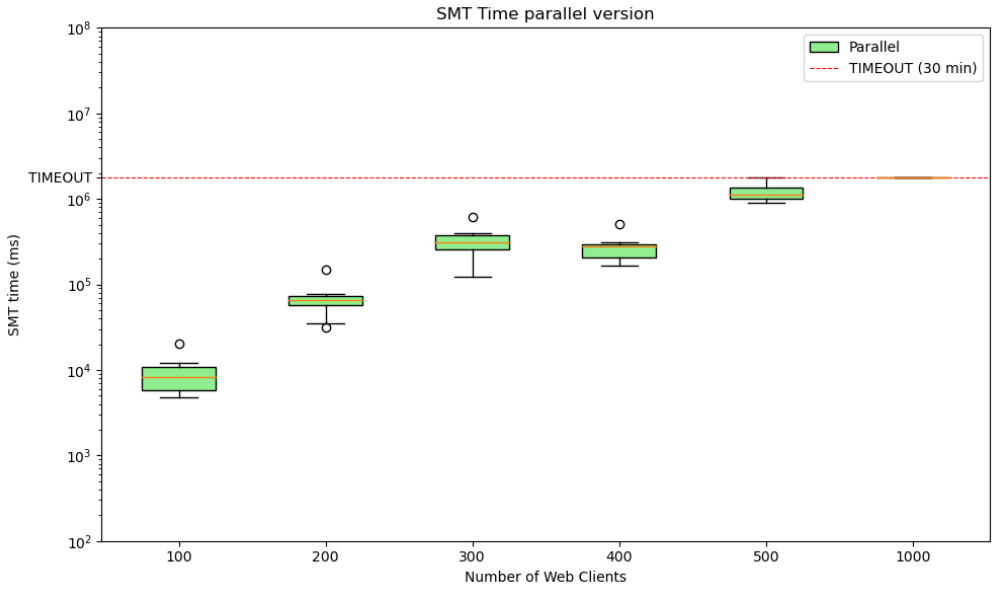

Figure 6.19. SMT time parallel version for large networks with Maximal Flows (GEANT)

6.2.2 VPNConfB networks performace

For all the tests on VPNConfB networks, the parameters of the Leiden algorithm were set as follows: resolution parameter = 1.0, MinNodes = 5, Normalization = "A". These values were chosen based on the guidelines established in the previous section to ensure a good clustering.

In this case the two versions of VEREFOO were compared on networks with dimension up to 200 webclients for Atomic Flows and up to 50 webclients for Maximal Flows. Again, the number of policies has been computed as $numberWebclients \times 3$. The results are shown in Fig. 6.20 for Atomic Flows and in Fig. 6.21 for Maximal Flows. With this network topology, the parallelized version shows a significant performance improvement for Atomic Flows, while for Maximal Flows the performance improvement is visible up to 20 webclients, then both versions fail to provide a solution before the timeout threshold of 30 minutes.

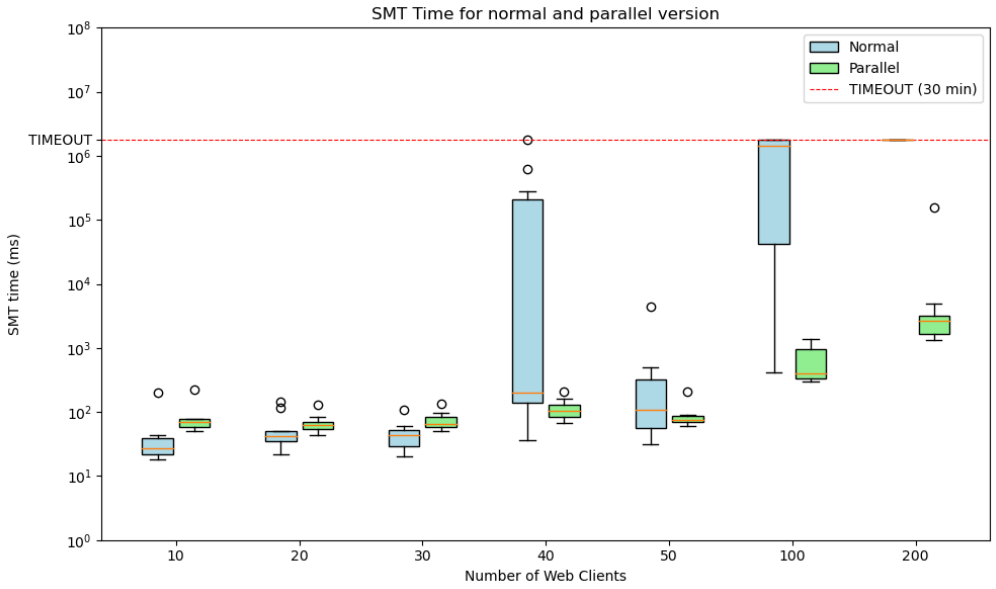

Figure 6.20. SMT time comparison standard vs parallel version with Atomic Flows (VPNConfB)

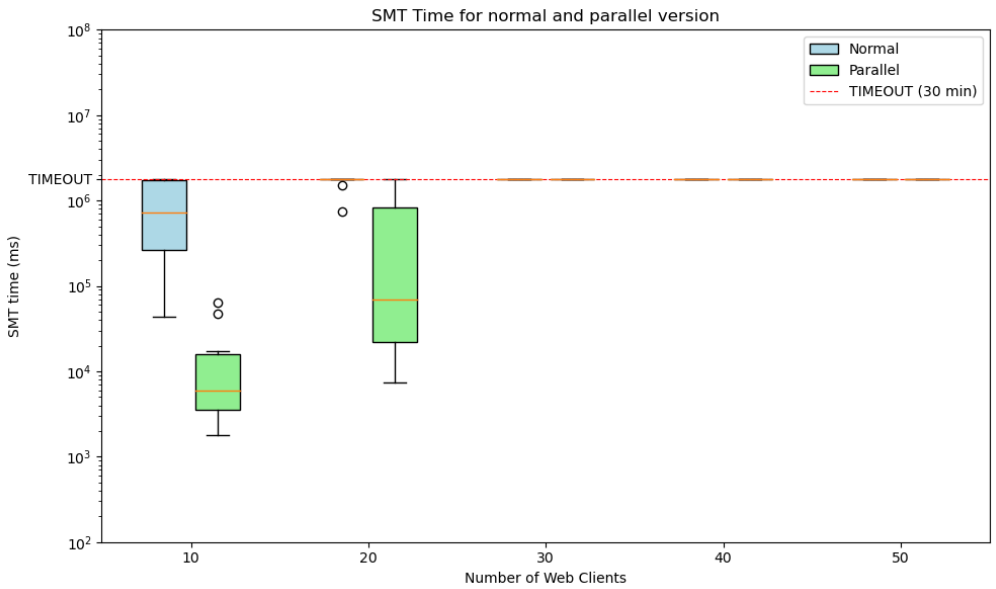

Figure 6.21. SMT time comparison standard vs parallel version with Maximal Flows (VPNConfB)

The optimality of the two solutions is compared in Fig. 6.22 and 6.23 for Atomic Flows and in Fig. 6.24 and 6.25 for Maximal Flows. Again, the optimality of the parallelized version is lower than the standard version, but the difference is smaller than in the GEANT networks.

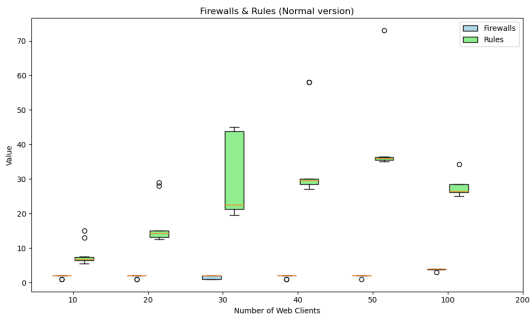

Figure 6.22. Firewalls and rules normal version with Atomic Flows (VPNConfB)

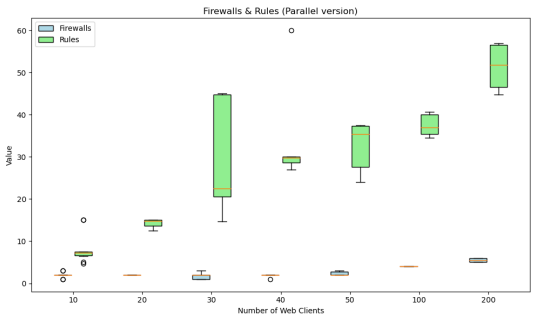

Figure 6.23. Firewalls and rules parallel version with Atomic Flows (VPNConfB)

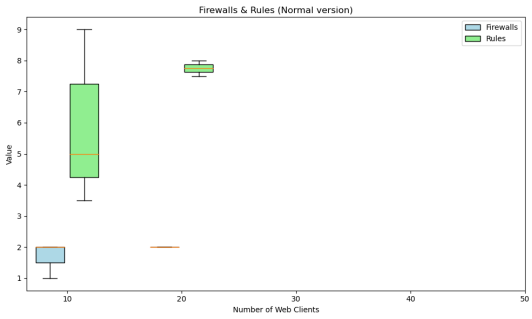

Figure 6.24. Firewalls and rules normal version with Maximal Flows (VPNConfB)

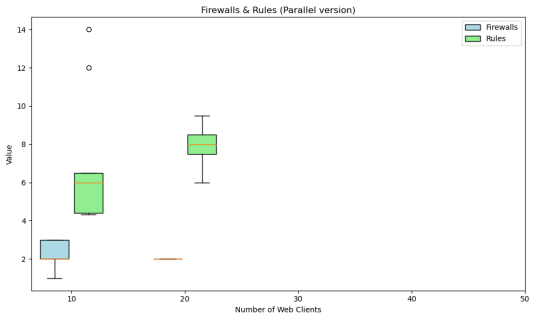

Figure 6.25. Firewalls and rules parallel version with Maximal Flows (VPNConfB)

Also in this case the parallel version has been tested on networks up to 1000 webclients. Since using Maximal Flows the maximum network size that can be solved within the timeout limit is 20 webclients, these tests were conducted only for Atomic Flows. The results are shown in Fig. 6.26, demonstrating that the parallel version can efficiently handle networks with up to 1000 webclients within a reasonable time frame.

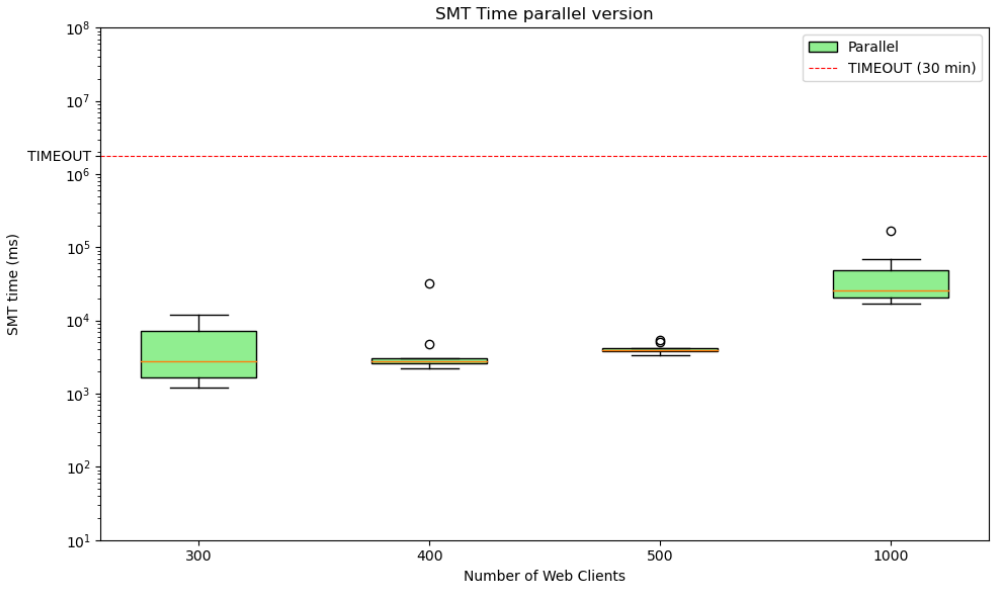

Figure 6.26. SMT time parallel version for large networks with Atomic Flows (VPNConfB)

Chapter 7

Conclusions

This thesis specifically focused on the VEREFOO tool, which uses formal methods to generate an optimal firewall configuration. Even if VEREFOO is automatic, avoids human errors and guarantees the optimal solution given a network and set of Network Security Requirements , its main limitation is scalability. It has been demonstrated that when applied to large-scale networks, the computational time required to find a solution becomes impractical. To address this, a novel approach was proposed and implemented, to partition a large network into smaller, more manageable clusters. By dividing the problem into smaller subproblems, VEREFOO was able to be applied to each cluster in parallel.

The results of this thesis show that this parallel approach significantly improves the performance of the standard version of VEREFOO. By processing the network in smaller, independent clusters, the overall time required to find a solution was drastically reduced. The challenges introduced by this division were also addressed, particularly the handling of "cutoff flows" that traverse multiple clusters. Specific strategies were devised to manage these flows, and a merging algorithm was created to resolve conflicts that arise when multiple clusters attempt to place a firewall in the same location. This method effectively balances the need for computational efficiency with the goal of maintaining a correct and near-optimal overall solution. First the parallel approach has been compared with the standard version showing a great performance improvement, then the parallel version was tested on bigger networks demonstrating it can handle networks composed of thousands of nodes.

In essence, this thesis successfully provided a scalable solution for VEREFOO, transforming it from a theoretical tool for small-scale networks into a practical solution for larger, real-world environments.

While the proposed parallel approach represents a significant step forward in terms of scalability, several areas can be improved in future research.

The current approach relies on the Leiden algorithm for network partitioning. Although effective, the Leiden algorithm has two main limitations. First, it requires the user to manually tune three input parameters, which can be a time-consuming and non-trivial task. The optimal parameter values can vary significantly between different network topologies, meaning the user must experiment to achieve the best results. Second, in certain network configurations, the Leiden algorithm may produce unbalanced clusters, where some clusters are significantly larger or contain

more flows than others. This unbalance can reduce the efficiency of the parallel approach, as the total runtime will be limited by the performance of the largest and most complex cluster.

Therefore, a key area for future work would be to replace the Leiden algorithm with a new, ad-hoc algorithm specifically designed for this purpose. The ideal algorithm would be completely automatic, requiring no parameters in input from the user, and would aim to create clusters that are as balanced as possible in terms of the number of traffic flows that must be managed. This would not only simplify the process for the user but also maximize the performance gains of the parallelization strategy.
Another possible improvement could be to the design and implementation of post-processing algorithms to remove redundant firewalls and rules in the final solution to increase the optimality of the solution. Finally, this approach may be extended to also to REACT-VEREFOO, a different version of VEREFOO that adds support for the reconfiguration of existing firewalls, enabling the reuse of previous configurations instead of forcing a full system re-deployment from scratch

Bibliography

[1] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," in *ACM Computing Surveys, Volume 56, Issue 3*, 2023. [Online]. Available: https://doi.org/10.1145/3616401

[2] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," in *IEEE Transactions on Dependable and Secure Computing*, 2024. [Online]. Available: https://doi.org/10.1109/TDSC.2024.3495230

[3] ——, "A two-fold traffic flow model for network security management," in *IEEE Transactions on Network and Service Management*, 2024. [Online]. Available: https://doi.org/10.1109/TNSM.2024.3407159

[4] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," in *IEEE Transactions on Dependable and Secure Computing*, 2022. [Online]. Available: https://doi.org/10.1109/TDSC.2022.3160293

[5] D. Bringhenti, R. Sisto, and F. Valenza, "Automating vpn configuration in computer networks," in *IEEE Transactions on Dependable and Secure Computing*, 2024. [Online]. Available: https://doi.org/10.1109/TDSC.2024.3409073

[6] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Automatic and optimized firewall reconfiguration," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10575212

[7] D. Bringhenti, F. Pizzato, R. Sisto, and F. Valenza, "A looping process for cyberattack mitigation," in *Proceedings of 2024 IEEE International Conference on Cyber Security and Resilience*, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10679501

[8] ——, "Autonomous attack mitigation through firewall reconfiguration," 2025. [Online]. Available: https://doi.org/10.1002/nem.2307

[9] V. traag, L. Waltman, and N. van Eck, "From louvain to leiden: guaranteeing well-connected communities," 2019. [Online]. Available: https://doi.org/10.1038/s41598-019-41695-z

[10] N. J. van Eck. (2023) Cwtsleiden/networkanalysis. [Online]. Available: https://github.com/CWTSLeiden/networkanalysis

[11] C. Basile, A. Lioy, C. Pitscheider, F. Valenza, and M. Vallini, "A novel approach for integrating security policy enforcement with dynamic network virtualization," in *Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015, London, United Kingdom, April 13-17, 2015*,

2015, pp. 1–5. [Online]. Available: https://doi.org/10.1109/NETSOFT.2015.7116152

[12] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool, "*Firmato*: A novel firewall management toolkit," *ACM Trans. Comput. Syst.*, vol. 22, no. 4, pp. 381–420, 2004. [Online]. Available: https://doi.org/10.1145/1035582.1035583

[13] J. M. Halpern and C. Pignataro, "Service function chaining (SFC) architecture," *RFC*, vol. 7665, pp. 1–32, 2015. [Online]. Available: https://doi.org/10.17487/RFC7665

[14] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for policy-based management," *RFC*, vol. 3198, pp. 1–21, 2001. [Online]. Available: https://doi.org/10.17487/RFC3198

[15] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model - version 1 specification," *RFC*, vol. 3060, pp. 1–100, 2001. [Online]. Available: https://doi.org/10.17487/RFC3060

[16] A. Matheus, "How to declare access control policies for XML structured information objects using oasis' extensible access control markup language (XACML)," in *38th Hawaii International Conference on System Sciences (HICSS-38 2005), CD-ROM / Abstracts Proceedings, 3-6 January 2005, Big Island, HI, USA*, 2005. [Online]. Available: https://doi.org/10.1109/HICSS.2005.300

[17] F. Valenza and A. Lioy, "User-oriented network security policy specification," *J. Internet Serv. Inf. Secur.*, vol. 8, no. 2, pp. 33–47, 2018. [Online]. Available: https://doi.org/10.22667/JISIS.2018.05.31.033

[18] J. D. Moffett and M. S. Sloman, "Policy hierarchies for distributed systems management," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1404–1414, Dec 1993.

[19] J. Zhou and J. Alves-Foss, "Security policy refinement and enforcement for the design of multi-level secure systems," *Journal of Computer Security*, vol. 16, no. 2, pp. 107–131, 2008. [Online]. Available: http://content.iospress.com/articles/journal-of-computer-security/jcs300

[20] A. K. Bandara, E. Lupu, J. D. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), 7-9 June 2004, Yorktown Heights, NY, USA*, 2004, pp. 229–239. [Online]. Available: https://doi.org/10.1109/POLICY.2004.1309175

[21] "Z3 programming guide," http://theory.stanford.edu/~nikolaj/programmingz3.html, accessed: 2019-04-29.

[22] "Verigraph repository," https://github.com/netgroup-polito/verigraph, accessed: 2019-04-29.

[23] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, "Formal verification of virtual network function graphs in an sp-devops context," in *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, 2015, pp. 253–262. [Online]. Available: https://doi.org/10.1007/978-3-319-24072-5_18

[24] B. E. Carpenter and S. W. Brim, "Middleboxes: Taxonomy and issues," *RFC*, vol. 3234, pp. 1–27, 2002. [Online]. Available: https://doi.org/10.17487/RFC3234

[25] D. Kreutz, F. M. V. Ramos, P. J. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015. [Online]. Available: https://doi.org/10.1109/JPROC.2014.2371999

[26] R. Chayapathi, S. F. Hassan, and P. Shah, *Network Functions Virtualization (NFV) with a Touch of SDN*. Addison-Wesley Professional, 2016.

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

[28] N. B. Youssef and A. Bouhoula, "A fully automatic approach for fixing firewall misconfigurations," in *11th IEEE International Conference on Computer and Information Technology, CIT 2011, Pafos, Cyprus, 31 August-2 September 2011*, 2011, pp. 461–466. [Online]. Available: https://doi.org/10.1109/CIT.2011.84

[29] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[30] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu, "Automatically repairing network control planes using an abstract representation," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, 2017, pp. 359–373. [Online]. Available: https://doi.org/10.1145/3132747.3132753

[31] K. Adi, L. Hamza, and L. Pene, "Automatic security policy enforcement in computer systems," *Computers & Security*, vol. 73, pp. 156–171, 2018. [Online]. Available: https://doi.org/10.1016/j.cose.2017.10.012

[32] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–7. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110402

[33] D. Bringhenti and F. Valenza, "Greenshield: Optimizing firewall configuration for sustainable networks," in *IEEE Transactions on Network and Service Management*, 2024. [Online]. Available: https://doi.org/10.1109/TNSM.2024.3452150

[34] D. Bringhenti, R. Sisto, and F. Valenza, "A novel abstraction for security configuration in virtual networks," 2023. [Online]. Available: https://doi.org/10.1016/j.comnet.2023.109745