Politecnico di Torino

Master of Science in Computer Engineering

Master Degree Thesis

Automatic security reaction in a virtualized environment

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

Candidate
Lorenzo Giglio

Academic Year 2023-2024

Summary

In the ever-evolving landscape of network security management, there's a pronounced shift from traditional manual systems to more advanced automated methods. This move isn't just a trend; it's a response to the complexities we face in the modern cybersecurity environment. Automated systems provide the benefits of faster responses, reducing the time window for potential breaches. They also offer greater consistency, minimizing human errors, which are a main concern in today's detailed and intricate cybersecurity scenarios. As threats evolve, the reliability and efficiency of these automated tools become increasingly essential. [1]

At the heart of this thesis is an exploration and extension of the VEREFOO (VErified REFinement and Optimized Orchestrator) framework [2] [3]. VEREFOO is designed to automate the daunting task of configuring packet filtering firewalls within a virtualized network environment [4]. It does so by converting high-level Network Security Requirements into optimized configurations. This is made possible by treating the challenge of firewall configuration as a MaxSMT problem, integrating optimization with formal verification processes.

The main motivation behind this research was to harness Intrusion Detection Systems alerts as real-time input for VEREFOO. In the face of a cyber attack, these alerts would guide the system in reconfiguring the network to fortify its defenses. A dedicated parser, developed as part of the research, stood at the forefront of this endeavor. Its main function is to efficiently process alerts generating from various Intrusion Detection Systems and transform them into Network Security Requirements, in a format compatible with VEREFOO.

One of the most important stages in the research focused on the development of a method that could seamlessly integrate new security requirements, drawn from alerts, into the pre-existing Network Security Requirements. This innovation ensured that the system could swiftly adapt and respond to emerging cyber threats. To execute these rapid adjustments in the real-world network, the integration of the virtual network translator module into VEREFOO was essential. This module interprets the Firewall Allocation Scheme generated by VEREFOO and translates it into a series of actionable files. Once prepared, these files become the foundation upon which the virtual network is initialized. The approach makes sure that the most current security measures are in place while also ensuring the operational continuity of the network.

Towards the end of this research, emphasis was placed on refining the automated response process to cyber threats. The thesis introduced a pivotal change by integrating React-VEREFOO into the established system. This adaptation was

significant; while the standard VEREFOO necessitates generating a Firewall Allocation Scheme from the ground up every time there's a change, React-VEREFOO offers a more agile approach. It capitalizes on existing configurations, allowing the network to be reconfigured with minimal redundancy, thereby conserving computational effort and enhancing efficiency.

The culmination of this study is the full automation of the response to cyber threats. The VEREFOO Log Integrator (vlogi) was developed to continuously monitor logs from Intrusion Detection Systems. When an alert is logged, vlogi coordinates with VEREFOO, React-VEREFOO, and VIP to initiate and deploy an updated network configuration, thus addressing the threat.

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my academic supervisors: Prof. Sisto, Prof. Valenza, and particularly Prof. Bringhenti, who consistently provided prompt and insightful responses to my queries. Their guidance has been invaluable throughout the development of my thesis.

I am deeply grateful to my family for their unwavering belief in me and their constant encouragement, which has been a pillar of strength, especially during challenging times. Their faith in my abilities has been a beacon of hope in this academic endeavor.

To the wonderful individuals I've been fortunate to meet during my time at Collegio Einaudi and now call friends, I cannot express enough gratitude. Your camaraderie, shared experiences, and the memories we've created have made this journey truly memorable.

When the night has come
And the land is dark
And the moon is the only light we'll see
No, I won't be afraid
Oh, I won't be afraid
Just as long as you stand
Stand by me

So darlin', darlin'
Stand by me, oh, stand by me
Oh, stand, stand by me
Stand by me

If the sky that we look upon
Should tumble and fall
Or the mountain should crumble to the sea
I won't cry, I won't cry
No, I won't shed a tear
Just as long as you stand
Stand by me

And darlin', darlin'
Stand by me, oh, stand by me
Oh, stand now, stand by me
Stand by me

Darlin', darlin'
Stand by me, oh, stand by me
Oh, stand now, stand by me
Stand by me
Whenever you're in trouble
Won't you stand by me?
Oh, stand by me
Won't you stand now?
Oh, stand, stand by me

Contents

List of Figures

Listings

Chapter 1

Introduction

Software-Defined Networking (SDN) has emerged as a transformative approach in modern network design and management. Unlike traditional networking paradigms which are constrained by static configurations and hardware limitations, SDN introduces a more fluid architecture in which the network control is decoupled from the forwarding plane, allowing centralized management. This promotes enhanced flexibility, scalability, and adaptability to changing application requirements. Running parallel to the evolution of SDN is the concept of Network Function Virtualization (NFV). NFV decouples network functions from proprietary hardware appliances, enabling them to run as software instances. Together, SDN and NFV work synergistically, with SDN providing the centralized control over networks and NFV offering the ability to deploy and run network services more flexibly and efficiently. [5]

As our reliance on digital systems grows, so does the importance of network security. Intrusion Detection Systems (IDSs), vital components in the cybersecurity world, continuously monitor network traffic for abnormal patterns. When such patterns are identified, IDSs generate alerts to notify network administrators of potential threats. This real-time surveillance system is essential for quick threat detection and mitigation, especially given the intricate and evolving nature of cyber-attacks.

The inherent dynamism of SDNs, amplified by the continuous stream of alerts from IDSs, underscores the need for an adaptive security mechanism. The main goal is to embed real-time network reconfiguration capabilities to counter detected threats automatically. By automating this process, it's possible to enhance response times, reduce risks associated with human errors, and ensure the network's resilience against a variety of threats.

The use of SDN, NFV, and IDSs presents a promising direction for cybersecurity. By centralizing control with SDN, virtualizing network functions through NFV, and maintaining vigilant monitoring via IDSs, networks can be more adaptive, responsive, and secure.

This research aims to establish a seamless method of integrating IDS alerts with SDN configurations. By doing so, it aspires to create an automated and real-time defense mechanism against cyber threats. A comprehensive breakdown

of objectives, thesis description, and contributions will be detailed in subsequent chapters.

The approach is structured as follows:

- **Firewall Allocation Scheme (FAS) generation**: Using VEREFOO, a Firewall Allocation Scheme (FAS) is generated from a file that encompasses both the network topology and its Network Security Requirements (NSRs).
- **Virtual network initiation**: With the FAS at hand, the necessary files to initiate a virtual network are generated.
- **Attack simulation**: An artificial cyber-attack is simulated to test the network's resilience.
- **Extraction and merging of NSRs**: Post-attack, the IDS (in this case, Snort 3) generates the corresponding alerts. From these alerts, new NSRs are extracted, which are then merged with the NSRs from the original file.
- **Regeneration of FAS**: With the combined file (topology + merged NSRs), a new FAS is generated using VEREFOO.
- **Virtual network update**: Finally, the running virtual network is updated using the files generated from the new FAS.

This research was conducted using Snort 3 and OSSEC 3.7 for intrusion detection and Docker to deploy the virtual network. It's essential to note certain limitations:

- **mergeRequirements algorithm**: This algorithm is still in its developmental stages. Its operation might not yet be at peak efficiency, and it requires further validation. Moreover, the algorithm currently makes specific assumptions about the network and the NSRs which may not be universally applicable.
- **Intrusion Detection Systems**: At present, the approach has been tailored specifically for Snort 3 and OSSEC 3.7, and adapting it for other IDSs may require some modifications.

The chapters are organized as follows:

- **Chapter 2: Background and related work**
  This chapter provides foundational knowledge of the key technologies and methodologies applied in this research. Firstly, the concept of Intrusion Detection Systems is discussed in detail. Within this context, two specific IDSs, Snort 3 and OSSEC 3.7, are introduced. An overview of both systems is presented, detailing their functionalities and their relevance to this research. A deep dive into VEREFOO provides insights into its various aspects, including Service and Allocation graphs, and Network Security Requirements. The latter part of this chapter offers a review of the existing literature and methodologies concerning VEREFOO, introducing the virtual network translator module and React-VEREFOO.

- **Chapter 3: Thesis overview**
  Before plunging into the intricate details, this chapter presents a concise layout of the entire thesis. It defines the objectives guiding the research and underscores the contributions made through this work.

- **Chapter 4: Approach**
  This chapter outlines the high-level procedure designed to automate our virtual network's response to cyber threats. It starts by detailing the modifications made to the standard VEREFOO demo topology and proceeds to guide on setting up the virtual network. The subsequent sections present a cyclic methodology for handling cyber threats, from recognizing an attack to automatically adjusting the network's security settings and validating the effectiveness of those adjustments. The closing section discusses the main observations and key takeaways derived from this method.

- **Chapter 5: Algorithm formulation**
  This chapter discusses the softwares developed or extended in the course of this research, starting with the VEREFOO IDS/IPS Parser (VIP). The spotlight then shifts to the virtual network translator algorithm and its integration with VEREFOO. The chapter concludes with a description of modifications made to React-VEREFOO, aimed at ensuring its compatibility with the developed attack response approach. Following these modifications, the chapter presents the revised approach, which now incorporates React-VEREFOO.

- **Chapter 6: Process automation**
  In this chapter, the mergeRequirements algorithm is explored, examining its key stages: preprocessing, merging, and postprocessing. The discussion includes an analysis of the algorithm's underlying assumptions and its limitations. Following this, the chapter introduces the vlogi tool, which incorporates the mergeRequirements algorithm and automates the entire attack response workflow.

- **Chapter 7: Demo validation**
  This chapter is dedicated to translating theory into practice. It demonstrates the developed approach by guiding the reader through the entire process. This includes generating the Firewall Allocation Scheme, launching the virtual network, simulating attacks, employing response automation through the vlogi tool, and verifying the updated NSRs.

- **Chapter 8: Conclusions and future work**
  This chapter draws together the primary outcomes and significant findings of the research. It underscores the achievements and simultaneously acknowledges the limitations and challenges faced in the course of the study. The latter part of the chapter suggests future research opportunities and potential enhancements based on the groundwork laid in this thesis.

Chapter 2

Background and related work

2.1 Intrusion Detection Systems

The progression of modern cyber threats calls for ever-evolving countermeasures. Central to these counteractive strategies are Intrusion Detection Systems, with a specific spotlight on rule-based ones. Such systems monitor network activities based on a well-defined set of rules to identify malicious or undesirable traffic.

Rules are built upon known patterns, signatures, or behaviors of potential cyber threats. Each rule helps the IDS identify certain types of harmful activities or unusual patterns in network traffic, offering a clear and organized way for the system to monitor the data passing through the network. When incoming traffic matches one of these rules, the IDS generates an alert, indicating a potential security breach or malicious activity.

The effectiveness of a rule-based IDS largely depends on the quality and relevance of its rules. Creating these rules requires an extensive understanding of network protocols, potential vulnerabilities, and common attack vectors. A carefully defined rule ensures that the IDS accurately identifies genuine threats, thereby reducing the number of false positives.

The cybersecurity domain is continuously changing, with new threats emerging and old ones evolving. This dynamism necessitates regular updates to the rule sets of IDSs to ensure they remain relevant and effective. Rule-based IDSs can adapt to these changes by incorporating new rules that reflect the current threat landscape, and by updating or removing outdated ones.

As previously mentioned, for this research, the role of Intrusion Detection Systems is fundamental, specifically the integration of Snort 3 and OSSEC 3.7. These are open-source rule-based systems known for their reliability and effectiveness in threat detection.

To bridge the gap between the IDS alerts from Snort 3 and OSSEC 3.7 and the VEREFOO framework, the VEREFOO IDS/IPS Parser was developed. The parser's primary function is to extract Network Security Requirements from the alerts generated by these IDSs. The ultimate objective is to leverage this information to automate the response to potential threats in a software-defined network.

The following subsections of this thesis will provide an in-depth look at Snort 3 and OSSEC 3.7, exploring their functionalities, unique features, and how they fit into the broader research context.

2.1.1 Snort 3

Originating in the late 1990s, Snort has consistently been at the forefront of intrusion detection and prevention solutions. Over the decades, it has undergone significant development, with Snort 3 being its most advanced version. Designed with versatility in mind, Snort 3 is equipped to handle complex network architectures and traffic patterns.

Snort 3 operates by analyzing network packets in real-time. Upon detecting suspicious activity, it generates alerts to notify administrators. One of the standout features of Snort 3 is its rule-driven architecture. This allows for customization and fine-tuning, ensuring that the detection mechanisms are tailored to the unique needs and nuances of individual network environments. This flexible, rule-based approach allows users to specify the characteristics of potential threats, making the detection process both precise and adaptable.

Several factors influenced the decision to integrate Snort 3. To begin with, Snort 3 has cultivated a strong reputation in the cybersecurity community over the years, known for its reliability and robustness. Furthermore, its open-source nature allows for transparent examination of its functionalities, making it more accessible and trustable. The simplicity of its design, combined with comprehensive documentation, means it is not just user-friendly but also fosters a deeper understanding of its inner workings. Its high customizability ensures that it can be tailored to meet specific requirements, making it adaptable to a variety of scenarios. Additionally, its capacity to easily interface with other systems enhances its applicability in diverse environments.

In the specific context of this thesis, Snort 3 is utilized as an exemplary Network-based Intrusion Detection System (NIDS) within our virtual network. Its implementation serves to demonstrate the applicability of our approach using a NIDS, since the methodologies developed can effectively integrate with different types of intrusion detection systems, including both network-based and host-based systems.

2.1.2 OSSEC 3.7

OSSEC, which stands for Open Source HIDS SECurity, is a comprehensive and widely used host-based Intrusion Detection System. Being a host-based IDS means it primarily focuses on analyzing the internal dynamics of a system rather than its network traffic. It provides log analysis, file integrity checking, policy monitoring, rootkit detection, and real-time alerting functionalities.

Over the years, OSSEC has witnessed multiple revisions and enhancements. OSSEC 3.7, as of the time of this writing, represents the latest version of this platform.

In pursuit of the primary research objective to automate responses to cyber threats based on alerts from Intrusion Detection Systems, the integration of diverse, robust systems was essential. Consequently, OSSEC has been incorporated as a representative example of Host-based Intrusion Detection Systems during the research phase. The selection of multiple IDSs emphasizes the research's capability to handle a variety of alert types and to apply the response mechanisms accordingly. OSSEC contributes valuable host-based detection insights, enriching the data pool for automated processing and response.

The integration of OSSEC is a practical step in this research. It wasn't chosen just to demonstrate the capabilities of the parser but to start building a system that can work with various types of IDSs. The VEREFOO IDS/IPS Parser is designed to handle alerts from different sources, including Snort 3 and OSSEC 3.7, paving the way for future updates that could allow for more complex threat response strategies.

2.2 VEREFOO

2.2.1 Overview

Software-Defined Networking and Network Functions Virtualization are making significant advancements, transforming network services by promising increased flexibility. These emerging technologies aim to bring about a change where manual, and often error-prone, configurations can be substituted with more dynamic and automated processes.

SDN and NFV, as fresh advancements in the domain of networking, are envisioned to revolutionize the way networking is approached. SDN enables the determination of traffic paths via software processes, while NFV offers the potential of virtualized network functions, all residing on general-purpose servers. With these technologies, service designers can conceptualize intended network services via Service Graphs (SGs), which depict the service functions and their interconnections.

Given the rapid virtualization of networks, we're witnessing an increasing feasibility in security automation. This is primarily due to the agility intrinsic to a virtualized setting, and the full control that software-based solutions provide over each network component. However, the automation of security defenses is still in its infancy. Among the numerous security tasks, the placement and configuration of Network Security Functions (NSFs) stand out as particularly daunting. NSFs are crucial as they are designed to meet Network Security Requirements, which represent the security constraints that a network's behavior should adhere to. The introduction of an automatic approach to this task not only conserves human effort but also assures optimal solutions that are provably correct.

VEREFOO emerges as a framework specifically crafted to function within these modern networking paradigms, enhancing the overall network management experience. Designed as a comprehensive solution to these challenges, VEREFOO aims to refine high-level NSRs, strategically allocate and configure selected NSFs [6], and enable the placement of each virtual function of the Service Graph on dedicated

servers within a physical substrate network. VEREFOO frames the problem internally as a partial weighted Maximum Satisfiability Modulo Theories (MaxSMT) problem. By doing so, VEREFOO guarantees both the formal correctness of its solutions and their optimality [7] [8].

To provide a clearer view, the following is a breakdown of VEREFOO's architecture (refer to Figure 2.1) and its essential modules, categorized into user inputs and processing modules.

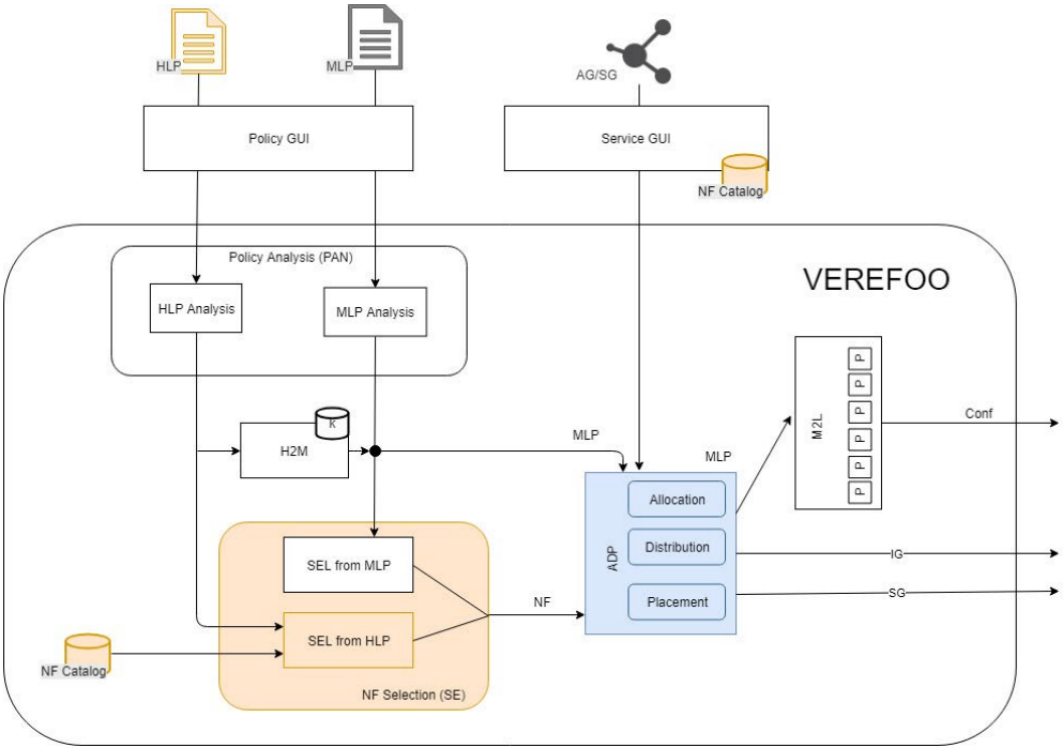

Figure 2.1. VEREFOO architecture

User inputs

- **Network Security Requirements**: Through a user-friendly Policy GUI, the service designer interacts with the framework to formulate security constraints. Depending on the expertise of the designer, these requirements can be expressed in either high-level or medium-level language.

- **Service Graph and Allocation Graph**: The Service GUI serves as an access point to the Network Functions Catalogue, offering the designer the flexibility to allocate functions directly onto their graph. This GUI allows either a Service Graph or an Allocation Graph to be input.

Processing modules

- **Policy ANalysis (PAN) module**: The PAN initiates the processing phase. It analyzes the NSRs for potential conflicts or errors. This module can either yield a minimal set of essential constraints or produce a comprehensive conflict report when automatic conflict resolution isn't feasible.
- **High-to-Medium (H2M) module**: When NSRs are framed in high-level language, the H2M module intervenes to refine these specifications into medium-level requirements. These refined requirements encapsulate essential data for the subsequent automatic policy creation for the NSFs and their respective configurations within the network.
- **NF Selection (SE) module**: This module identifies the requisite NSFs to meet the NSRs. Using a pre-existing catalogue (identical to the one accessible through the Service GUI), it selects the necessary functions [9].
- **Allocation, Distribution and Placement (ADP) module**: Undoubtedly, the ADP module is a fundamental part of VEREFOO's architecture. Tasked with the delivery of the final graph equipped with allocated and configured NSFs, it uses the *z3* theorem prover to handle the aforementioned partial weighted MaxSMT problem. It's worth noting that while NSRs are treated as non-negotiable hard constraints, supplementary specifications and optimizations are managed as weighted soft constraints.

2.2.2 Service Graph

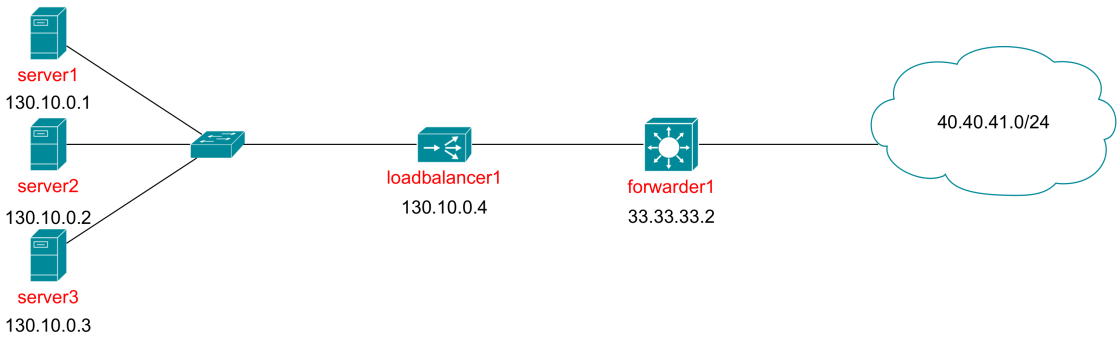

Figure 2.2. Visual representation of an example Service Graph

The Service Graph (SG) holds a crucial role in the operation of the VEREFOO framework, serving as a comprehensive representation of the network topology upon which Network Security Requirements are based. [10] Fundamentally, the SG is a logical topology of a virtual network, showcasing the interconnections between various service functions and network nodes to form a complete end-to-end network service.

The Service Graph incorporates complex architectures with multiple traffic paths, diverging from the linear arrangements found in Service Function Chains (SFCs), and thus, represents a generalization of the SFC concept. By allowing a varied arrangement of service functions and network nodes, the SG provides a comprehensive description of the network topology. This is achieved while assuming the correct implementation of lower-level functions such as switches and routers, which are crucial for directing incoming packets based on forwarding or routing tables, even though they are not explicitly included in the SG.

The task of defining a Service Graph falls upon the network service designer, who selects from a variety of Network Functions (NFs) to create the SG. These NFs could range from web caching to load balancing, with an emphasis on their forwarding behavior as opposed to their complete behavior. In simpler terms, the primary concern is how a traffic flow is forwarded and altered by an NF, rather than the minutiae of the function's operations [11].

Figure 2.2 illustrates a condensed representation of the network topology discussed in this thesis. This layout features a cluster of web servers positioned behind a load balancer. The load balancer is interconnected via a forwarder to a network populated with web clients.

Now, let's examine the XML representation of the aforementioned Service Graph:

Listing 2.1. XML representation of an example Service Graph

```
<graph id="0" serviceGraph="true">
    <node functional_type="WEBSERVER" name="130.10.0.1">
        <neighbour name="130.10.0.4" />
        <configuration description="e1" name="httpserver1">
            <webserver>
                <name>130.10.0.1</name>
            </webserver>
        </configuration>
    </node>
    <node functional_type="WEBSERVER" name="130.10.0.2">
        <neighbour name="130.10.0.4" />
        <configuration description="e2" name="httpserver2">
            <webserver>
                <name>130.10.0.2</name>
            </webserver>
        </configuration>
    </node>
    <node functional_type="WEBSERVER" name="130.10.0.3">
        <neighbour name="130.10.0.4" />
        <configuration description="e3" name="httpserver3">
            <webserver>
                <name>130.10.0.3</name>
            </webserver>
        </configuration>
    </node>
    <node functional_type="LOADBALANCER" name="130.10.0.4">
        <neighbour name="130.10.0.1" />
        <neighbour name="130.10.0.2" />
        <neighbour name="130.10.0.3" />
        <neighbour name="33.33.33.2" />
```

```
        <configuration description="s9" name="loadbalancer">
            <loadbalancer>
                <pool>130.10.0.1</pool>
                <pool>130.10.0.2</pool>
                <pool>130.10.0.3</pool>
            </loadbalancer>
        </configuration>
    </node>
    <node functional_type="FORWARDER" name="33.33.33.2">
        <neighbour name="130.10.0.4" />
        <neighbour name="40.40.41.-1" />
        <configuration name="ForwardConf">
            <forwarder>
                <name>Forwarder</name>
            </forwarder>
        </configuration>
    </node>
    <node functional_type="WEBCLIENT" name="40.40.41.-1">
        <neighbour name="33.33.33.2" />
        <configuration description="e4" name="officeA">
            <webclient nameWebServer="130.10.0.1" />
        </configuration>
    </node>
</graph>
```

- The graph element has a unique identifier and a boolean attribute called *serviceGraph*. A `true` value indicates a Service Graph, while `false` denotes an Allocation Graph (default), as both utilize the same representation.

- Each network entity is represented as a *node*. The unique identifier of a *node* is its IP address, specified by the *name* attribute.

- Every *node* is associated with a specific role or function within the network, denoted by the *functional_type* attribute. This attribute defines the *node*'s primary responsibility, such as serving web content (`WEBSERVER`), acting as a loadbalancer (`LOADBALANCER`), forwarding packets (`FORWARDER`), or acting as a web client (`WEBCLIENT`).

- A *node*'s immediate connections or neighbors are listed using the *neighbour* elements, each containing a *name* attribute specifying the IP address of the neighboring node. These relationships map out the network's interconnections.

- The *configuration* element provides additional metadata about a node. Its structure and contents can vary based on the *node*'s function. For instance, in the case of the `LOADBALANCER`, the configuration element lists the IP addresses of all servers in its balancing pool, fundamental for VEREFOO to correctly and optimally generate firewall rules.

2.2.3 Allocation Graph

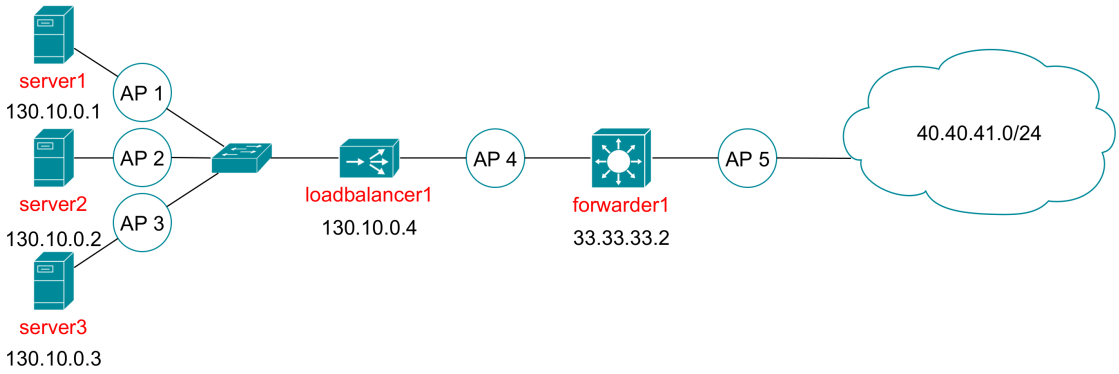

Figure 2.3. Visual representation of an example Allocation Graph

If a Service Graph is provided, VEREFOO proceeds to transform it into an internal representation known as the Allocation Graph (AG). This process includes the generation of Allocation Places (APs) for each link between network nodes or functions, creating potential positions for firewall placement.

A seasoned service designer can directly build the AG, either by mandating the placement of a firewall at a specific AP or by excluding certain APs as valid positions for firewalls. This user input introduces a level of flexibility and can narrow down the solution space that VEREFOO must explore, effectively reducing computation times. However, it is important to acknowledge that this manual intervention could also lead to issues, such as the inability to find a solution or the derivation of a suboptimal solution, particularly if potentially optimal positions are disregarded based on user specifications.

Now, let's examine the XML representation of the aforementioned Allocation Graph:

Listing 2.2. XML representation of an example Allocation Graph

```
<graph id="0">
    <node functional_type="WEBSERVER" name="130.10.0.1">
        <neighbour name="1.0.0.1" />
        <configuration description="e1" name="httpserver1">
            <webserver>
                <name>130.10.0.1</name>
            </webserver>
        </configuration>
    </node>
    <node functional_type="WEBSERVER" name="130.10.0.2">
        <neighbour name="1.0.0.2" />
        <configuration description="e2" name="httpserver2">
            <webserver>
                <name>130.10.0.2</name>
            </webserver>
        </configuration>
    </node>
```

```
        <node functional_type="WEBSERVER" name="130.10.0.3">
            <neighbour name="1.0.0.3" />
            <configuration description="e3" name="httpserver3">
                <webserver>
                    <name>130.10.0.3</name>
                </webserver>
            </configuration>
        </node>
        <node name="1.0.0.1">
            <neighbour name="130.10.0.1" />
            <neighbour name="130.10.0.4" />
        </node>
        <node name="1.0.0.2">
            <neighbour name="130.10.0.2" />
            <neighbour name="130.10.0.4" />
        </node>
        <node name="1.0.0.3">
            <neighbour name="130.10.0.3" />
            <neighbour name="130.10.0.4" />
        </node>
        <node functional_type="LOADBALANCER" name="130.10.0.4">
            <neighbour name="1.0.0.1" />
            <neighbour name="1.0.0.2" />
            <neighbour name="1.0.0.3" />
            <neighbour name="1.0.0.4" />
            <configuration description="s9" name="loadbalancer">
                <loadbalancer>
                    <pool>130.10.0.1</pool>
                    <pool>130.10.0.2</pool>
                    <pool>130.10.0.3</pool>
                </loadbalancer>
            </configuration>
        </node>
        <node name="1.0.0.4">
            <neighbour name="130.10.0.4" />
            <neighbour name="33.33.33.2" />
        </node>
        <node functional_type="FORWARDER" name="33.33.33.2">
            <neighbour name="1.0.0.4" />
            <neighbour name="1.0.0.5" />
            <configuration name="ForwardConf">
                <forwarder>
                    <name>Forwarder</name>
                </forwarder>
            </configuration>
        </node>
        <node name="1.0.0.5">
            <neighbour name="33.33.33.2" />
            <neighbour name="40.40.41.-1" />
        </node>
        <node functional_type="WEBCLIENT" name="40.40.41.-1">
            <neighbour name="1.0.0.5" />
            <configuration description="e4" name="officeA">
                <webclient nameWebServer="130.10.0.1" />
            </configuration>
        </node>
    </graph>
```

This Allocation Graph closely resembles the Service Graph we examined earlier. However, it also incorporates Allocation Places, which are the *node* elements within the `1.0.0.0/24` network that are missing both *functional_type* and *configuration*.

2.2.4 Network Security Requirements

VEREFOO provides four distinct approaches to design Network Security Requirements:

- *whitelisting*: A conservative approach, whitelisting starts with a default behavior that blocks all traffic flows. The designer's role is to specify which traffic flows are permitted. This approach is most suitable for networks where a minimal and specific set of communications need to be permitted.
- *blacklisting*: Contrary to whitelisting, the default behavior in this approach is to allow all traffic flows. The designer's role is to specify which traffic should be denied access.
- *specific*: In this approach, users can explicitly define reachability and isolation requirements. VEREFOO will then automatically decide whether to follow a *rule-oriented* or a *security-oriented* strategy for all other cases not explicitly addressed by the user:
  - *rule-oriented*: This strategy aims to minimize the number of rules.
  - *security-oriented*: This strategy focuses on permitting only essential communications necessary to meet all the user's specifications.

It's crucial to note that for the *specific* approach, the user-provided NSRs should be anomaly-free, meaning they shouldn't have conflicts or suboptimizations. While this might seem constraining, any anomalies can be addressed using established anomaly analysis techniques [12] [13].

Each Network Security Requirement is articulated using a mid-level language [14], enabling users to clearly define the specific IP 5-tuple of the allowed or disallowed traffic flows. This translates to each NSR comprising six distinct attributes:

- *ruleType*: Specifies the nature of the security requirement. Possible values are `ReachabilityProperty` or `IsolationProperty`.
- *IPSrc*: Defines the source IP address. A wildcard symbol (`-1`) may be used to denote entire subnetworks.
- *IPDst*: Identical in functionality to IPSrc, but for the destination IP address.
- *portSrc*: Indicates the source port. The wildcard symbol can be employed to signify all ports in the range `0-65535`.
- *portDst*: Identical in functionality to portSrc, but for the destination port.

- *transportProto*: Specifies the transport protocol. Permissible values include:
  - `TCP`: Transmission Control Protocol.
  - `UDP`: User Datagram Protocol.
  - `OTHER`: Any transport protocol excluding TCP/UDP.
  - `ANY`: Any transport protocol, inclusive of TCP/UDP.

Now, let's analyze an example list of NSRs:

Listing 2.3. Example list of Network Security Requirements

```
<PropertyDefinition>
    <!-- policy 1 -->
    <Property name="ReachabilityProperty" graph="0" src="130.10.0.4"
        dst="40.40.41.-1" lv4proto="ANY" />
    <!-- policy 2 -->
    <Property name="ReachabilityProperty" graph="0" src="40.40.41.-1"
        dst="130.10.0.4" lv4proto="TCP" dst_port="80" />
    <!-- policy 3 -->
    <Property name="IsolationProperty" graph="0" src="40.40.41.-1"
        dst="130.10.0.4" lv4proto="TCP" dst_port="0-79" />
    <Property name="IsolationProperty" graph="0" src="40.40.41.-1"
        dst="130.10.0.4" lv4proto="TCP" dst_port="81-65535" />
    <!-- policy 4 -->
    <Property name="IsolationProperty" graph="0" src="40.40.41.-1"
        dst="130.10.0.4" lv4proto="UDP" />
    <!-- policy 5 -->
    <Property name="IsolationProperty" graph="0" src="40.40.41.-1"
        dst="130.10.0.4" lv4proto="OTHER" />

    <!-- Repeat policies for all servers within the loadbalancer's pool -->
</PropertyDefinition>
```

In Listing 2.3, the *specific* approach is utilized to set up the policies. It blocks all traffic from `40.40.41.0/24` to `130.10.0.0/24`, except for the traffic on TCP port 80. On the other hand, any traffic from `130.10.0.0/24` to `40.40.41.0/24` is allowed. Following the *security-oriented* strategy, VEREFOO will block any other traffic that is not required in this scenario, specifically all UDP and OTHER traffic from `130.10.0.0/24` to `40.40.40.0/24`. This setup ensures that the only possible action in this network is for clients on the `40.40.41.0/24` network to access web pages from the servers in the `130.10.0.0/24` network using TCP port 80.

2.3 Prior work in VEREFOO

2.3.1 The virtual network translator module

The virtual network translator [15] is a module specifically developed for VEREFOO to convert a Firewall Allocation Scheme into a functional virtual network. This transformation involves converting the FAS into a series of actionable files that can be directly used to instantiate and operate the virtual network. Before

generating the virtual network files, this module processes VEREFOO's output, eliminating any superfluous Allocation Places and ensuring that only the necessary network nodes are retained. This creates an optimized environment for the subsequent translation process.

The responsibility of converting the refined FAS into actionable files – such as Docker Compose configurations, assorted configuration files, and Dockerfiles for various container types – falls to the `VnetworkTranslator` and `IptablesVnetwork` classes. By supporting a diverse range of functional types, the virtual network translator module allows users to design and implement intricate network configurations.

Presently, the module is equipped to only generate **iptables** configuration files. Moving forward, there are plans to enhance the module's functionality to support additional firewall software options, including EBPF and OpenvSwitch, aiming to increase its versatility and extend its applicability.

2.3.2 React-VEREFOO

React-VEREFOO [16] [17] represents a modified version of the VEREFOO framework, tailored to address specific limitations observed in the original implementation. In its existing form, VEREFOO necessitates a complete recomputation of firewall configurations every time changes are made to the set of Network Security Requirements. This process results in a substantial computational burden, with the time required ranging from some seconds to several minutes, depending on the complexity of the network.

React-VEREFOO emerges as a solution to this issue, aiming to significantly reduce the computation time needed for network reconfiguration in response to changes in NSRs. This tool is designed to generate a reconfigured version of the current settings in the shortest time possible, ensuring that the modifications do not disrupt the validity of existing NSRs.

Maintaining the formal approach of VEREFOO, React-VEREFOO guarantees the correctness of the configuration by construction, ensuring that any changes made do not compromise the integrity of the network's security policies.

In the context of this thesis, VEREFOO was initially employed to develop an approach capable of automatically responding to cyber threats through the reconfiguration of a virtual network. As highlighted earlier, VEREFOO operates by generating configurations from the ground up, which can be computationally intensive. To address this, React-VEREFOO was integrated into the system, resulting in increased performance and efficiency.

Chapter 3

Thesis overview

3.1 Thesis objectives

The primary aim of this thesis centered around devising a strategy to effectively utilize alerts from various Intrusion Detection Systems as a means to counteract cyber threats.

A subsidiary objective emerged from this central goal, necessitating the development of a parser capable of converting alerts generated by diverse IDSs into Network Security Requirements that are compatible with the VEREFOO framework.

Further branching from this, it was essential to devise a method for integrating the newly extracted requirements with the pre-existing set, ensuring a cohesive and effective update process.

Building on these foundational objectives, the next logical step involved conceptualizing a comprehensive methodology to serve as the framework for updating a network's security configuration in real-time, allowing for a swift and calculated response to cyber attacks.

The integration of React-VEREFOO into the workflow emerged as a natural progression to enhance the efficiency of the developed methodology. As highlighted previously, VEREFOO operates by completely reevaluating and regenerating configurations for any change in Network Security Requirements. In contrast, React-VEREFOO brings a level of efficiency to the process, utilizing the existing Firewall Allocation Scheme, which contains the old configurations. This allows for a more rapid reconfiguration process, as the system only needs to adjust the necessary components according to the updated Network Security Requirements.

Lastly, the development of the VEREFOO Log Integrator (vlogi) represents the culmination of these objectives. vlogi acts as an orchestrator, using VEREFOO, React-VEREFOO, and VIP to update the network's configuration in response to ongoing cyber attacks. This integration signifies the achievement of a fully automated, efficient, and responsive system for network security management.

3.2 Contributions

The contributions of this thesis significantly enhance the capabilities of the VEREFOO framework, setting a foundation for future advancements in automated cyber attack response:

- **VEREFOO IDS/IPS Parser (VIP)**: This component is responsible for translating alerts from various Intrusion Detection Systems into Network Security Requirements that can be interpreted by the VEREFOO framework. While its current implementation is compatible with only two IDSs and two alert formats, its design is modular and extensible, allowing for straightforward integration of additional formats and IDSs as needed.
- **Integration of the virtual network translator module**: This module has been integrated into the VEREFOO framework and expanded to accommodate the translation of traffic monitor nodes, in addition to its existing capabilities.
- **mergeRequirements algorithm**: An algorithm has been developed to combine the initial requirements with those extracted by the VIP tool.
- **Cyclical cyber attack response methodology**: Drawing on the tools and modules detailed above, a cyclical methodology was formulated to autonomously respond to cyber threats by updating security configurations. This methodology was subsequently implemented as a script, serving as a tangible proof of concept within the VEREFOO framework.
- **Integration of React-VEREFOO**: To strengthen efficiency, React-VEREFOO has been incorporated into the workflow, substantially reducing the computational time necessary for reconfigurations.
- **VEREFOO Log Integrator (vlogi)**: vlogi functions as a central orchestrator within the network security framework. It continuously monitors logs from various Intrusion Detection Systems. Upon identifying an alert, vlogi coordinates the network's response by engaging with VEREFOO, React-VEREFOO, and VIP. It dynamically updates the network's security configurations in response to the detected threats, thereby ensuring a rapid and precise defense mechanism. Similarly to the VIP tool, vlogi is designed with modularity and extensibility in mind, facilitating the inclusion of preliminary log processing, support for various types of firewall software, and the incorporation of new functionalities as needed.

Chapter 4

Approach

4.1 Adapting the official demo topology

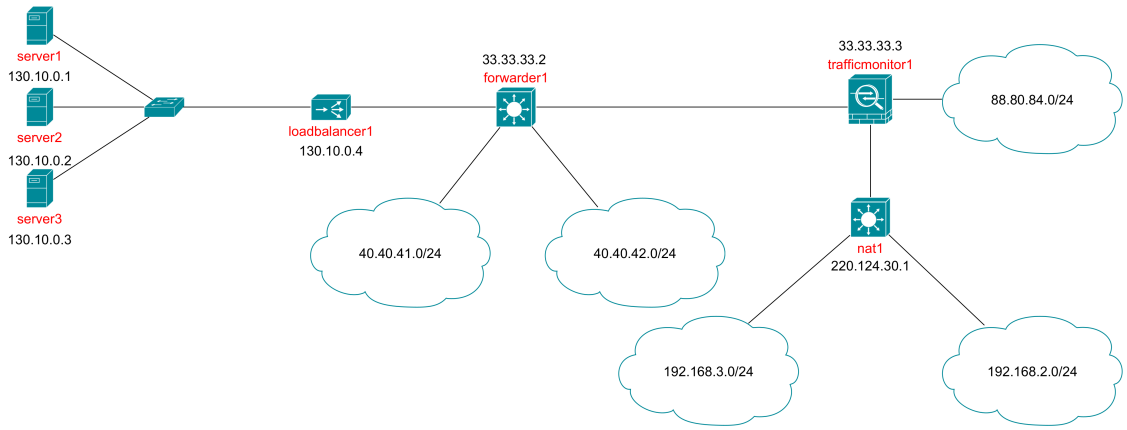

Figure 4.1. Network topology used in this thesis

This work employs VEREFOO's demo network topology as a base, with specific modifications tailored to our requirements.

The primary alteration involves reconfiguring the traffic monitor node. In the original demo topology, this node was designated as a simple forwarder without explicit mention of the IDS software. The relevant section in the original configuration was as follows:

Listing 4.1. Traffic monitor definition inside original network topology

```
<node functional_type="FORWARDER" name="33.33.33.3">
    <neighbour name="1.0.0.7" />
    <neighbour name="1.0.0.8" />
    <neighbour name="1.0.0.9" />
    <configuration name="ForwardConf">
        <forwarder>
            <name>Forwarder</name>
        </forwarder>
```

```
    </configuration>
</node>
```

In our adapted topology, we redefine this node's functional type to `TRAFFIC_MONITOR` and specify the IDS software within the inner `configuration` node. The new configuration is:

Listing 4.2. Traffic monitor definition inside new network topology

```
<node functional_type="TRAFFIC_MONITOR" name="33.33.33.3">
    <neighbour name="1.0.0.7" />
    <neighbour name="1.0.0.8" />
    <neighbour name="1.0.0.9" />
    <configuration name="trafficmonitor1">
        <traffic_monitor>
            <name>snort3</name>
        </traffic_monitor>
    </configuration>
</node>
```

This adjustment necessitated a minor update to the virtual network translator in order to incorporate support for traffic monitors.

As a prerequisite for testing our network against a cyber attack, it was essential to implement one of Snort's community rules. The chosen rule is designed to detect a login attempt performed on a machine which was infected by the QAZ worm. The rule is stated as follows:

Listing 4.3. Snort community rule

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 7597 (
    msg:"MALWARE-BACKDOOR QAZ Worm Client Login access";
    flow:to_server,established; content:"qazwsx.hsq";
    metadata:ruleset community; classtype:misc-activity; sid:108;
    rev:12; )
```

This rule is particularly suitable for our purposes, as its activation requires only a simple action: establishing a connection to **TCP port 7597** and sending a message that includes *qazwsx.hsq*.

We designated the host at `192.168.3.1` as the attacker and the `130.10.0.0/16` network as the victim.

The original demo topology prohibited traffic on TCP port 7597 from the `192.168.3.0/24` to the `130.10.0.0/24` network. To accommodate our test scenario, we made necessary adjustments to policies **3**, **4**, and **5**.

The original policies were as follows:

Listing 4.4. Original demo policies

```
<!-- policy 3 -->
<Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
    dst="130.10.0.4" dst_port="80" lv4proto="TCP"/>
<!-- policy 4 -->
<Property graph="0" name="IsolationProperty" src="192.168.3.-1"
    dst="130.10.0.4" dst_port="0-79" lv4proto="TCP" />
```

```
<Property graph="0" name="IsolationProperty" src="192.168.3.-1"
    dst="130.10.0.4" dst_port="81-65535" lv4proto="TCP"/>
<!-- policy 5 -->
<Property graph="0" name="IsolationProperty" src="192.168.3.-1"
    dst="130.10.0.4" lv4proto="UDP" />

<!-- Repeat policies for all servers within the loadbalancer's pool -->
```

Our modified policies are as follows:

Listing 4.5. New demo policies

```
<!-- policy 3 -->
<Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
    dst="130.10.0.4" lv4proto="TCP" />
<!-- policy 4 -->
<Property graph="0" name="IsolationProperty" src="192.168.3.-1"
    dst="130.10.0.4" lv4proto="UDP" />
<!-- policy 5 -->
<Property graph="0" name="IsolationProperty" src="192.168.3.-1"
    dst="130.10.0.4" lv4proto="OTHER" />

<!-- Repeat policies for all servers within the loadbalancer's pool -->
```

These changes enable all TCP traffic from `192.168.3.0/24` to reach `130.10.0.0/24`, regardless of the destination port, while still blocking other types of traffic.

4.2 Setup and demonstration

This section offers an overview of the approach outlined in Chapter 1, encompassing the setup and execution of the demonstration.

4.2.1 Setting up and launching the virtual network

The initial phase involves providing VEREFOO with an XML file that delineates our virtual network. This file is used to generate the Firewall Allocation Scheme.

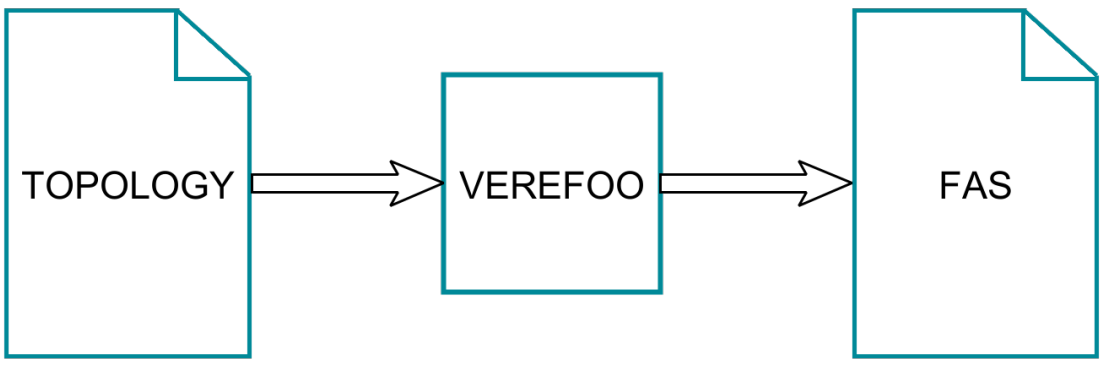

Figure 4.2. Step 1 of 7

Subsequently, we use the FAS to create the necessary files, such as docker-compose and configuration scripts, for deploying our virtual network.

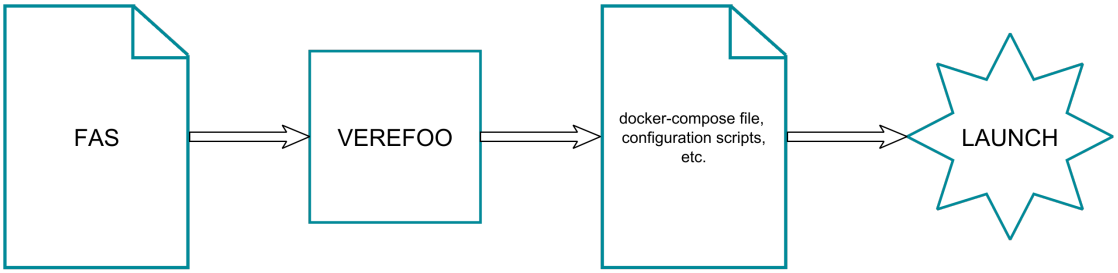

Figure 4.3. Step 2 of 7

4.2.2 Attack simulation

Once our network is operational, we conduct a simulated cyber attack. For this scenario, we assume that our servers have been compromised by the QAZ worm, which opens a backdoor on **TCP port 7597**. An unauthorized client (`client7`), seeking to exploit this backdoor for a Denial-of-Service attack, sends a login request to `loadbalancer1`, which routes the request to one of the compromised servers. Upon a successful login attempt through this backdoor, our traffic monitor (`trafficmonitor1`) should identify this suspicious activity and generate an alert.

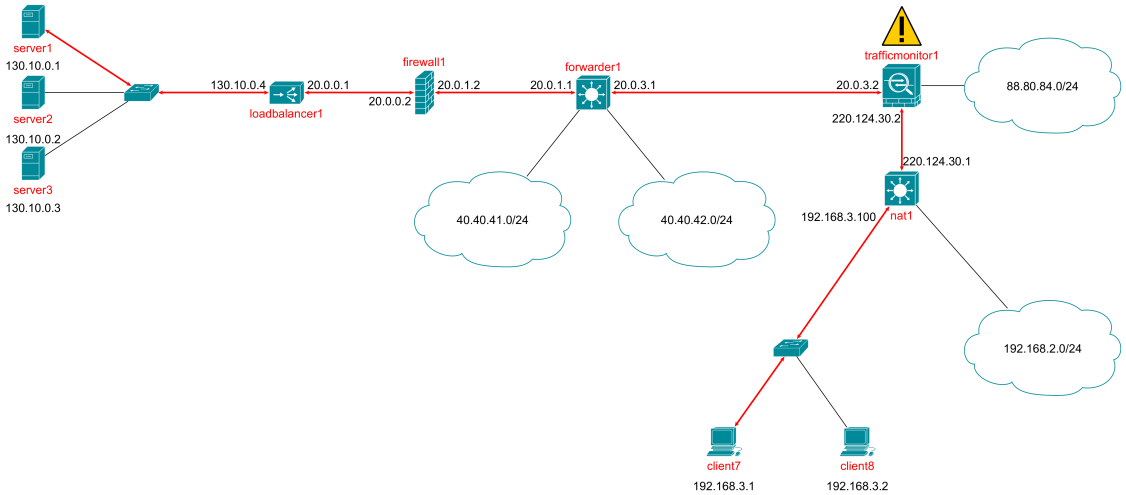

Figure 4.4. Step 3 of 7

4.2.3 Extracting new requirements and merging

The VIP tool evaluates the alert, extracts crucial information, and generates the corresponding VEREFOO `Property` nodes. The extracted requirements are then merged with those of the initial topology using the `mergeRequirements` script.

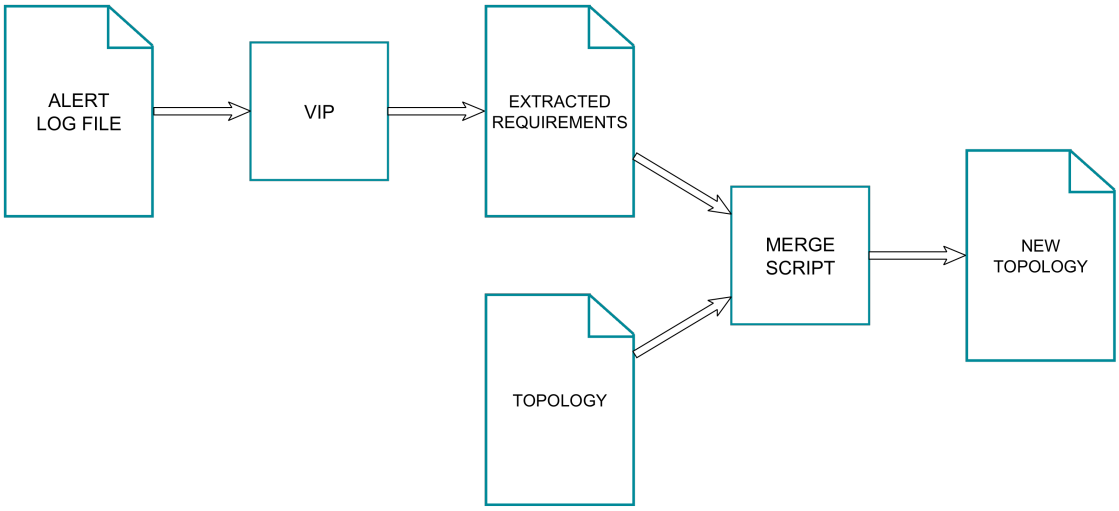

Figure 4.5. Step 4 of 7

4.2.4 Updating the Firewall Allocation Scheme

With the new requirements, we generate an updated FAS, akin to the process in **Step 1** (Figure 4.2).

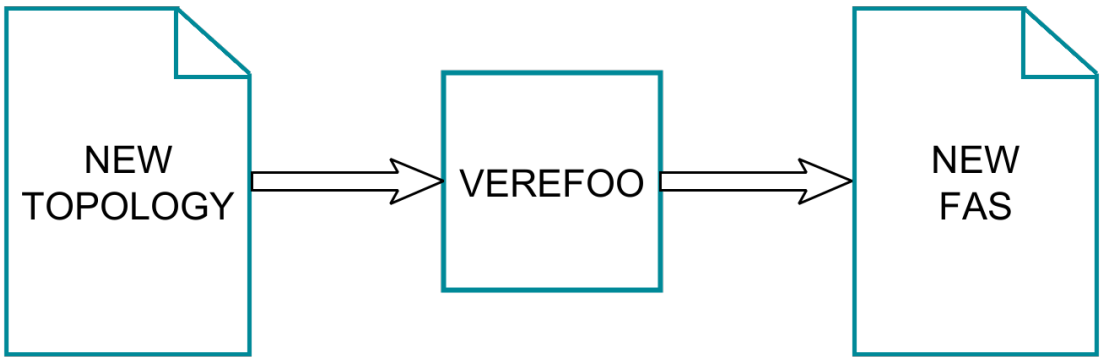

Figure 4.6. Step 5 of 7

Following that, we prepare the necessary files for deploying the new virtual network, similar to **Step 2** (Figure 4.3).

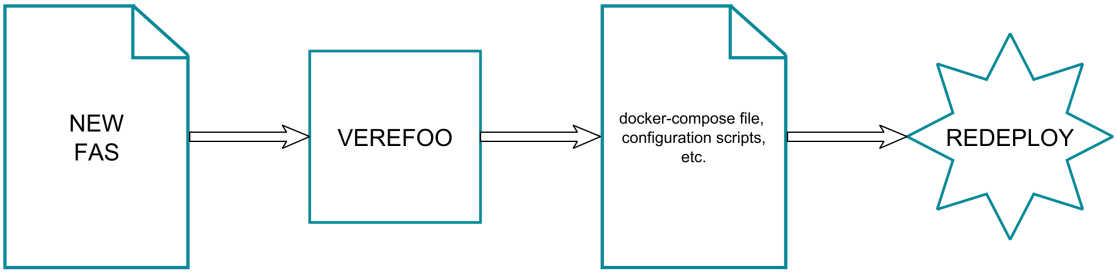

Figure 4.7. Step 6 of 7

4.2.5 Redeploying and verifying

Finally, to test the effectiveness of our changes, we simulate the attack again, following the same procedure as in **Step 3** (Figure 4.4). If the attack is successfully blocked, it indicates that our modifications have been effective.

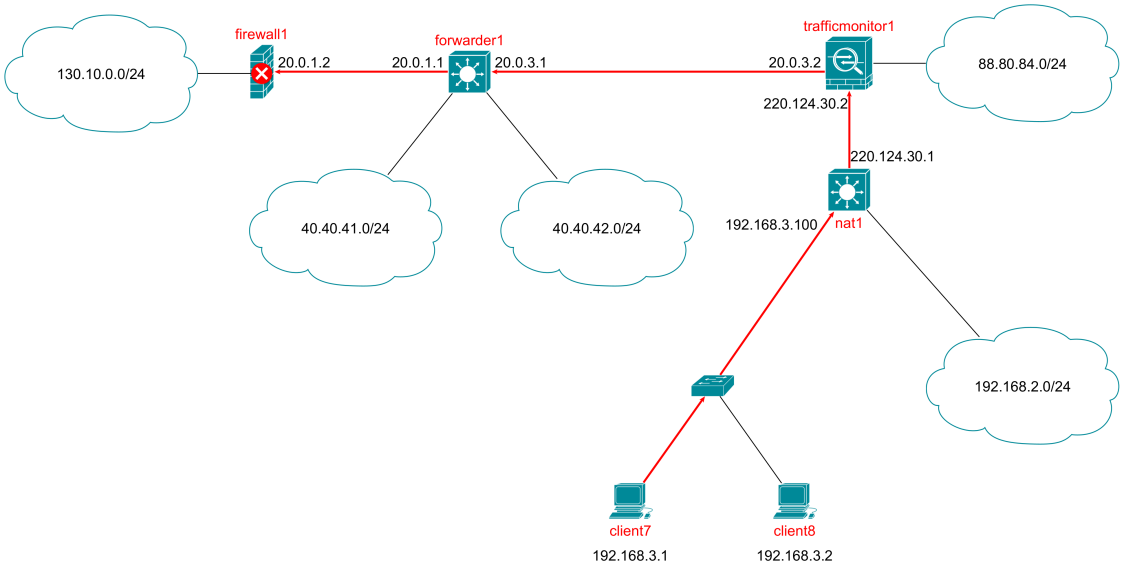

Figure 4.8. Step 7 of 7

4.3 Results and observations

The approach establishes a network with an inherent defense mechanism against cyber threats by utilizing the automated distribution of firewalls and rules provided by VEREFOO. This automated system is capable of detecting and responding to cyber threats without the need for manual intervention. Network configurations can be updated in real-time through modifications to configuration files. This process ensures that only specific components needing updates are affected, negating the necessity for a complete network restart.

Figure 4.9 shows the process cycle for network defense automation. The cycle starts with the generation of the Firewall Allocation Scheme, proceeds with the launch of the virtual network, and upon the detection of an attack, the system extracts and integrates the new requirements, which then leads to the regeneration of the FAS and redeployment of the virtual network.

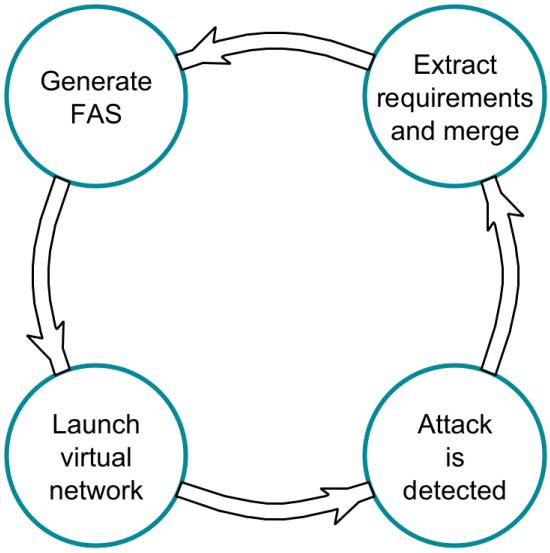

Figure 4.9. Attack response cycle

Chapter 5

Algorithm formulation

5.1 VEREFOO IDS/IPS Parser

The VEREFOO IDS/IPS Parser (VIP) is a Spring Boot application featuring REST APIs that are designed to parse alerts from potentially every IDS/IPS system. It efficiently translates these alerts into input requirements compatible with the VEREFOO framework, with a design that allows for easy expansion to accommodate additional IDS/IPS systems in the future.

The main goal of this section is to give a general overview of the architecture of VIP. It describes the roles and functions of each component, highlights how they work together, and provides guidance on how the software can be extended in the future.

5.1.1 Motivation and need

The development of VIP was motivated by a key challenge: each Intrusion Detection System generates alerts in its own unique format. These formats can vary widely, from plain text files to CSV files, JSON documents, and beyond. Although different IDSs can be configured to produce alerts in various formats, they all share common critical information, such as source IP, destination IP, source port, destination port, and protocol. Therefore, there was a clear need for a tool that could uniformly process this diverse range of alert formats, extracting all relevant information.

VIP addresses this need by providing a flexible solution. When a user submits a request, they specify the name of the IDS, its version, and the alert format in the request URL, and include the alerts in the body of the request. Based on this input, VIP selects the appropriate parser class to accurately extract the pertinent information from the alerts. The output is a list of `Property` nodes, encapsulated within a root `PropertyDefinition` node. In cases where the alert is malformed or there are issues with the URL and related parameters, VIP is designed to return an error message, ensuring clarity and accuracy in the parsing process.

This approach ensures that VIP can accommodate alerts from different IDS systems, converting them into a standardized format compatible with VEREFOO, thereby facilitating the subsequent stages of threat response.

5.1.2 Architectural overview

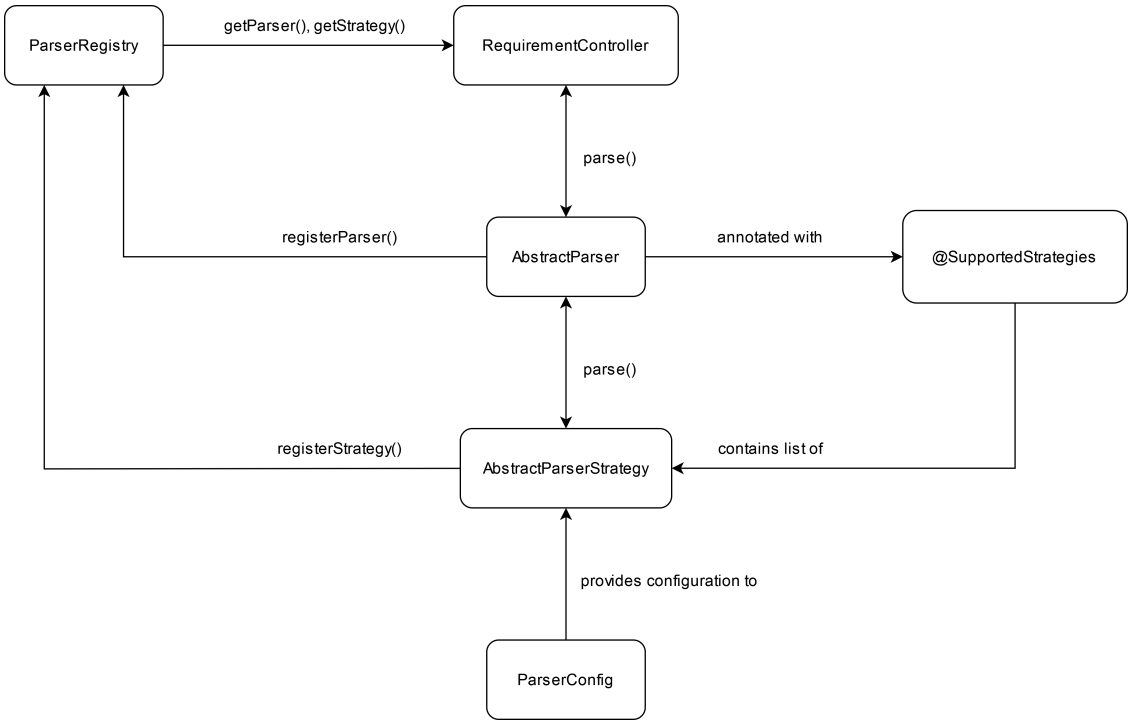

Figure 5.1. Diagram of VIP's main components and their interactions

Upon system startup, parsers derived from the `AbstractParser` class and strategies derived from the `AbstractParserStrategy` class register themselves within the `ParserRegistry`. This step is crucial for preparing the system to handle incoming parsing tasks by ensuring that all necessary components are readily available.

The `ParserRegistry` is a central component that maintains a list of all supported parsers and strategies, as well as a map that associates each parser with its compatible strategies. This structure is key to the system's ability to efficiently match parsing requests with the appropriate resources.

When a parsing request is received, the `RequirementController`'s `parseAlerts()` method is invoked. This method uses the path variables `idsName`, `idsVersion`, and `alertMode` to determine the correct parser and strategy to be used for the request by calling the `getParser()` and `getStrategy()` methods from the `ParserRegistry`.

The selected `AbstractParser`'s `parse()` method is then executed. This method processes the input stream containing the alerts and applies the specified alert priority and graph ID to the parsing operation. If these parameters are not provided, default values are used. The `parse()` method subsequently calls the chosen strategy's `parse()` method, providing it with the necessary parameters.

Parsing logic is contained within the `AbstractParserStrategy`'s `parse()` method. This method uses the configuration settings specified in the

`application.properties` file, which are retrieved through the `ParserConfig` class.

Additionally, parsers based on the `AbstractParser` class are annotated with `@SupportedStrategies`, indicating the strategies they support. This flexibility allows strategies to be reused with different IDS versions as needed.

Exception handling is managed through a dedicated package that contains custom exceptions tailored for the system. The `GlobalExceptionHandler` class ensures that any standard exception is handled gracefully, maintaining the stability and reliability of the system.

Example request

- `POST /api/parser/snort/3/AlertFastV0?priority=2`
- Request body:

Listing 5.1. Example request body with alerts generated by Snort 3

```
08/05-14:27:15.908164 [**] [1:2013504:6] ET TROJAN Observed
   Malicious SSL Cert (Likely Malware CnC Domain Related)
   [**] [Classification: A Network Trojan was Detected]
   [Priority: 1] {TCP} 192.168.1.2:56340 ->
   217.160.0.120:443
08/05-14:27:19.428771 [**] [1:2014819:5] ET SCAN Behavioral
   Unusual Port 22 traffic Potential Scan or Inbound Attack
   [**] [Classification: Detection of a Network Scan]
   [Priority: 2] {TCP} 92.118.37.80:56834 -> 192.168.1.2:22
08/05-14:30:20.534654 [**] [1:2001219:20] ET POLICY PE EXE
   or DLL Windows file download HTTP [**] [Classification:
   Potentially Bad Traffic] [Priority: 2] {TCP}
   192.168.1.2:56340 -> 104.20.1.85:80
08/05-14:32:45.835122 [**] [1:2010935:3] ET TROJAN ELF/Mirai
   Variant User-Agent (Inbound) [**] [Classification: A
   Network Trojan was Detected] [Priority: 1] {TCP}
   192.168.1.2:56340 -> 217.160.0.120:443
```

- Return value:

Listing 5.2. Extracted VEREFOO requirements

```
<PropertyDefinition>
   <Property graph="0" name="IsolationProperty" src="192.168.1.2"
   dst="217.160.0.120" dst_port="443" lv4proto="TCP"/>
   <Property graph="0" name="IsolationProperty" src="92.118.37.80"
      dst="192.168.1.2" dst_port="22" lv4proto="TCP"/>
   <Property graph="0" name="IsolationProperty" src="192.168.1.2"
      dst="104.20.1.85" dst_port="80" lv4proto="TCP"/>
</PropertyDefinition>
```

Custom exceptions

- 404 (Not Found)
  - **ParserNotFoundException**: Occurs when the `idsName` and `idsVersion` combination does not correspond to any existing parser.
  - **AlertModeNotFoundException**: Triggered when the specified `alertMode` in the request does not exist.
- 400 (Bad Request)
  - **MalformedAlertException**: Raised when the alert lacks required fields or has improper formatting, making information extraction impossible.
  - **ProtocolErrorException** (Snort 3-specific): The alert contains `Error` as its protocol.
  - **RequestPriorityException**: Occurs when the priority level specified in the request is outside the acceptable range for the chosen parser.
  - **AlertPriorityException**: Applies when the priority level within the alert itself falls outside the parser's acceptable range.
  - **InvalidIPsException**: Raised for source or destination IPs that are invalid or not properly formatted (e.g., `192.168.1.256`).
  - **InvalidPortNumbersException**: Indicates that the alert includes port numbers outside the valid range of 0 to 65535.
  - **UnexpectedPortsException**: Triggered when port information is supplied for a protocol that does not use ports.
  - **PortMismatchException**: Occurs when one of the ports is provided in the alert while the other is missing.
  - **SamePortTypeException**: Indicates that both ports in the alert are of the same type, either ephemeral or non-ephemeral.
  - **UnsupportedAlertModeException**: Raised when the `alertMode` specified does not match any supported mode for the given `idsName` and `idsVersion` combination.
  - **ConstraintViolationException**: Applies when any request parameter does not adhere to the specified constraints.
- 500 (Internal Server Error)
  - **RequestStreamException**: Indicates an error encountered while obtaining the input stream from the request.
  - **StreamProcessingException**: Triggered by issues in reading from the input stream.
  - **StrategyNotSetException**: Occurs when the parser's strategy has not been set before calling the `AbstractParserStrategy.parse()` method.

5.1.3 Main components

RequirementController

The RequirementController is responsible for managing the API endpoint `POST /api/parser/{idsName}/{idsVersion}/{alertMode}`. Upon receiving a request, the `parseAlerts()` method is called to process the request. It generates a `RequirementSet` which is then automatically transformed into an XML response. The response contains a `PropertyDefinition` root element that encapsulates all the `Property` nodes extracted from the alerts.

AbstractParser

The AbstractParser provides a general representation of a parser. Its main features include abstract methods that define the IDS name, version, and the range of priority levels it operates with. Each parser has a `defaultPriority` attribute, which is set upon instantiation by its child classes. It is also designed to manage variable priority level definitions, where the numeric value of the lowest priority may exceed that of the highest. The `shouldFilter` method accounts for such variations during alert processing. Furthermore, the `parse()` method, upon invocation, triggers the `parse()` method of the `AbstractParserStrategy` passed to it, provided that the strategy is not null.

AbstractParserStrategy

The AbstractParserStrategy encapsulates the generic logic needed to parse specific alert modes, such as the `alert_fast` mode in Snort. It maintains a reference to the `ParserConfig`, which is utilized by various helper methods within the strategy. These helper methods are responsible for validating the alert's consistency, formatting, etc. The central `parse()` method within AbstractParserStrategy contains the core parsing logic.

AbstractParserStrategy.ParsingContext

Located within the `AbstractParserStrategy`, the ParsingContext class stores details about the alert currently being processed. It acts as a central data repository, enabling the passage of the context itself rather than multiple individual parameters during the parsing process.

@SupportedStrategies

The @SupportedStrategies annotation is utilized on classes extending `AbstractParser`. It specifies which strategies are supported by a given parser, aiding in its configuration.

ParserConfig

ParserConfig is a configuration class that provides access to the parameters defined in the `application.properties` file under the `vip` prefix.

ParserRegistry

The ParserRegistry functions as a repository for all parsers and strategies. It maintains a list of available parsers, a list of strategies, and a map that links each parser with its corresponding strategies. Registration of parsers and strategies is conducted through the `registerParser()` and `registerStrategy()` methods. The `getParser()` method is used to retrieve a specific parser by requiring both `idsName` and `idsVersion`. Similarly, the `getStrategy()` method fetches a strategy given a supported parser and the alert mode in question.

5.1.4 Extending VIP

To add support for new Intrusion Detection Systems and their alert modes in VIP, simply follow the steps detailed below:

1. **Create a new parser class**
    - Within the `parser` package, create a subclass of `AbstractParser` corresponding to the specific IDS intended for support.
    - Annotate the newly created class with `@Component` to ensure it is recognized as a Spring-managed component.
    - Implement all abstract methods, which entails:
        - Defining a unique name and version for the parser. This is essential because for POST requests to `/api/parser/{idsName}/{idsVersion}/{alertMode}`, the `idsName` and `idsVersion` path variables must correspond to the parser's name and version to correctly identify and utilize the required parser class.
        - Establishing the range for the lowest and highest priority levels that the parser can handle.
        - Specifying a default priority within the constructor, which will be used in cases where no priority level is explicitly stated in the incoming request.
2. **Create a new strategy class**
    - Create one or more subclasses of `AbstractParserStrategy` within the `strategy` package, each corresponding to different alert modes of the target IDS.
    - Ensure that each new strategy class is given a distinct name to avoid conflicts.

- Annotate the strategy classes with `@Component`.
- Implement the `parse()` method. This method should utilize helper methods for alert validation and manage a `ParsingContext` instance, which should be updated as new information from the alerts is extracted and processed.

3. **Annotate the parser class** with `@SupportedStrategies`, providing a list of the strategies that the parser class supports, in the following manner: `@SupportedStrategies({"StrategyClass1", "StrategyClass2", ...})`.

As previously noted, for POST requests made to `/api/parser/{idsName}/{idsVersion}/{alertMode}`, the `idsName` and `idsVersion` are intended to match the name and version of the parser class, while `alertMode` should correspond to the name of the strategy class, with all matching being case-insensitive.

In scenarios where parsers for the same IDS share common functionalities, such as classes, attributes, or methods, it is advisable to consolidate these shared elements into a utility class. An example of this would be the `SnortUtils` class for parsers related to Snort.

5.2 Integrating and adapting the virtual network translator to support traffic monitors

The virtual network translator was initially designed to eliminate superfluous forwarders from VEREFOO's output and to append a `description` attribute to each `FIREWALL` node. This attribute aids in the unique identification of `FIREWALL` nodes by VEREFOO's deployer, responsible for creating configuration files compatible with various firewall software.

However, the translator had two main problems. Firstly, it prematurely removed "unnecessary" forwarders from the output. These forwarders, while deemed unnecessary for immediate translation, are actually critical for React-VEREFOO. They serve as potential Allocation Places for new firewalls, and their early removal could hinder React-VEREFOO from finding optimal or viable reconfiguration solutions. Therefore, a balance needed to be struck: keeping these forwarders for React-VEREFOO while removing them for translation purposes. Secondly, its translation process was not API-driven but was automatically initiated whenever VEREFOO generated a Firewall Allocation Scheme.

To address these issues, the framework's behavior was adjusted. Previously, VEREFOO would transform unused Allocation Places into `FORWARDER` nodes. This process was altered to convert these Allocation Places into `VFORWARDER` nodes (**V**EREFOO **FORWARDER**), facilitating the identification and subsequent removal of unnecessary forwarders prior to translation. The code responsible for removing forwarders was then relocated to precede the translator invocation during the API call. A more in-depth discussion on the introduction of `VFORWARDER` nodes is presented in the next section.

Additionally, a new API endpoint was created: `POST verefoo/adp/venvironment/generateFiles`, with a mandatory request parameter, `firewallType`. This parameter specifies the firewall software to be used by VEREFOO's deployer. As of the current writing, `firewallType` can only be `IPTABLES`.

Regarding traffic monitor adaptation, as previously mentioned, two different types of Intrusion Detection Systems were taken into consideration: Network-based IDSs and Host-based IDSs. From a forwarding perspective, NIDSs are analogous to standard `FORWARDER` nodes, making their integration relatively straightforward. The challenge lay in integrating HIDSs, which required a mechanism to denote the presence of a specific IDS within a node. This was achieved by introducing a new configuration element, `monitor_name`, in both `WEBSERVER` and `MAILSERVER` nodes:

Listing 5.3. Definition of `monitor_name`

```
<xsd:element name="monitor_name" type="xsd:string" nillable="true" />
```

The `monitor_name` element within the node's configuration specifies the IDS to be installed:

Listing 5.4. Example usage of `monitor_name` in server configuration

```
<node functional_type="WEBSERVER" name="130.10.0.1">
    <neighbour name="1.0.0.1" />
    <configuration description="e1" name="httpserver1">
        <webserver>
            <name>130.10.0.1</name>
            <monitor_name>ossec3.7local</monitor_name>
        </webserver>
    </configuration>
</node>
```

Supporting traffic monitors also entailed developing Dockerfiles for both network nodes with Snort 3 and servers with OSSEC 3.7 (local installation). This development included code for generating docker-compose files with appropriate start commands and volumes for log monitoring.

5.3 React-VEREFOO

React-VEREFOO is an enhancement of the original VEREFOO framework, designed to accelerate the Firewall Allocation Scheme generation process. It achieves this by utilizing results from previous runs to adapt the network according to new Network Security Requirements, rather than recalculating everything anew.

It adopts the Atomic Flows model for representing traffic flows. This model employs Atomic Predicates (APs), a concept introduced in 2015 as a solution for the Network Reachability problem [18] and later integrated into VEREFOO by prior research [19].

5.3.1 Limitations

In React-VEREFOO, `FORWARDER` nodes are one of the functional types subject to reconfiguration. However, this poses a challenge due to the dual nature of how these nodes are included in network topologies. On one hand, forwarders might be explicitly added by users in the input network topology, in which case they are not meant to be modified. On the other hand, VEREFOO can automatically insert `FORWARDER` nodes in place of unused Allocation Places. In these instances, the forwarders are considered eligible for reconfiguration.

Another issue in React-VEREFOO is encountered when managing multiple requirements based on several port ranges and protocols. This complexity leads to a rapid growth in the number of Atomic Flows computed, which in turn substantially increases the constraints input into the solver. This increase significantly complicates the task of finding viable solutions, often making it difficult to solve the problem within a reasonable timeframe.

5.3.2 Adapting React-VEREFOO to be used within the proposed methodology

To address the issue of React-VEREFOO incorrectly reconfiguring `FORWARDER` nodes added by users, a distinct functional type, `VFORWARDER`, was introduced. React-VEREFOO now utilizes this type to replace unused Allocation Places instead of standard `FORWARDER` nodes, ensuring that only these `VFORWARDER` nodes are subject to reconfiguration. Concurrently, modifications were made to the virtual network translator, enabling it to remove only `VFORWARDER` nodes during the translation process.

Initially, unused forwarders in VEREFOO were tagged as `unNeeded` in their configuration, simplifying their identification and removal during translation. However, with the integration of React-VEREFOO, the introduction of the `VFORWARDER` type became necessary to avoid disrupting existing tests and functionalities within the system. This solution represented the most straightforward and least disruptive modification required for the integration of React-VEREFOO within the approach.

5.3.3 Building new test networks

In response to the challenge of React-VEREFOO's extended timeframes in generating new FASs, a simplification of the original test network (Figure 4.1) was necessary. The network was reduced to its essential components while retaining key functionalities, ensuring an effective demonstration of the attack response mechanism.

As previously mentioned, our study focused on two distinct network configurations. The first network used Snort 3 as a network-based IDS, and the second employed OSSEC 3.7 as a host-based IDS, specifically installed on one of the servers.

For the network incorporating Snort 3 as a NIDS, the original topology was streamlined by removing all non-essential subnetworks, as illustrated in Figure 5.2.

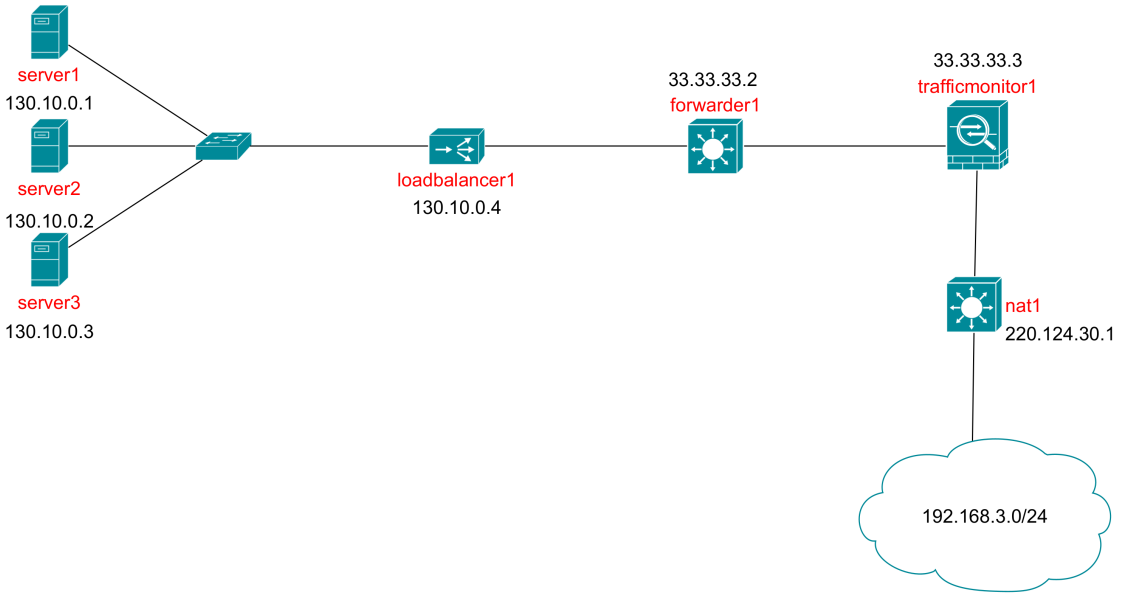

Figure 5.2. Test network with Snort 3 as NIDS

For the network featuring OSSEC 3.7 as a HIDS, the load balancer and NAT components were removed. This decision aimed at creating a simpler yet distinct test network, as depicted in Figure 5.3.

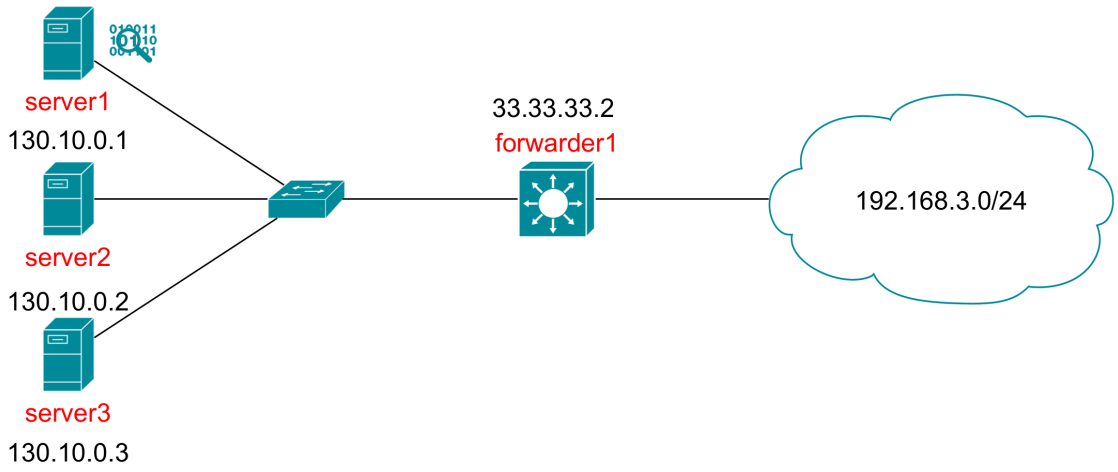

Figure 5.3. Test network with OSSEC 3.7 HIDS on `server1`

5.3.4 The updated approach

With the introduction of React-VEREFOO, our methodology requires some adjustments to accommodate this new element. This section revisits the steps outlined in Chapter 4 and details the necessary modifications. Changes are highlighted in red

for clarity. The fundamental structure remains largely unchanged, with adaptations made specifically for React-VEREFOO compatibility.

The initial step now integrates VEREFOO with newly added support for VFORWARDER nodes.

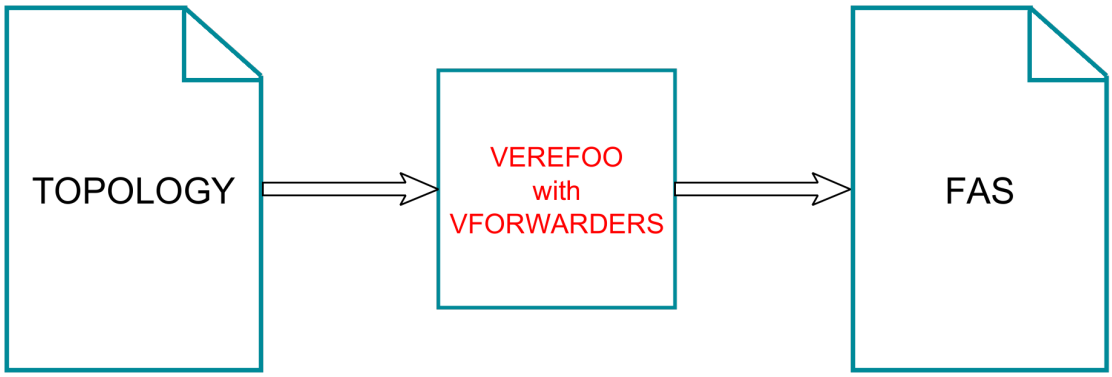

Figure 5.4. Step 1 of 7 (React-VEREFOO)

The second step involves a straightforward substitution of VEREFOO with React-VEREFOO for network file generation.

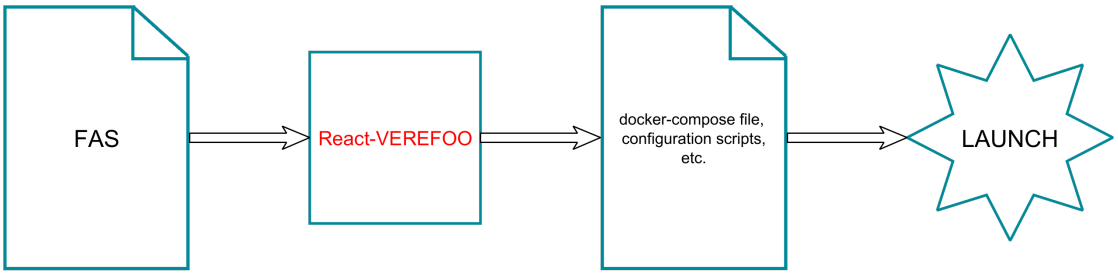

Figure 5.5. Step 2 of 7 (React-VEREFOO)

The third step is the same as before, since it simply involves conducting an attack.

Given that React-VEREFOO utilizes the existing FAS to reconfigure the network, requirements extracted from IDS alerts are merged with the previously generated FAS instead of the regular topology. The resulting requirements consist of a `PropertyDefinition` element encompassing the merged NSRs and an `InitialProperty` element containing the original NSRs.

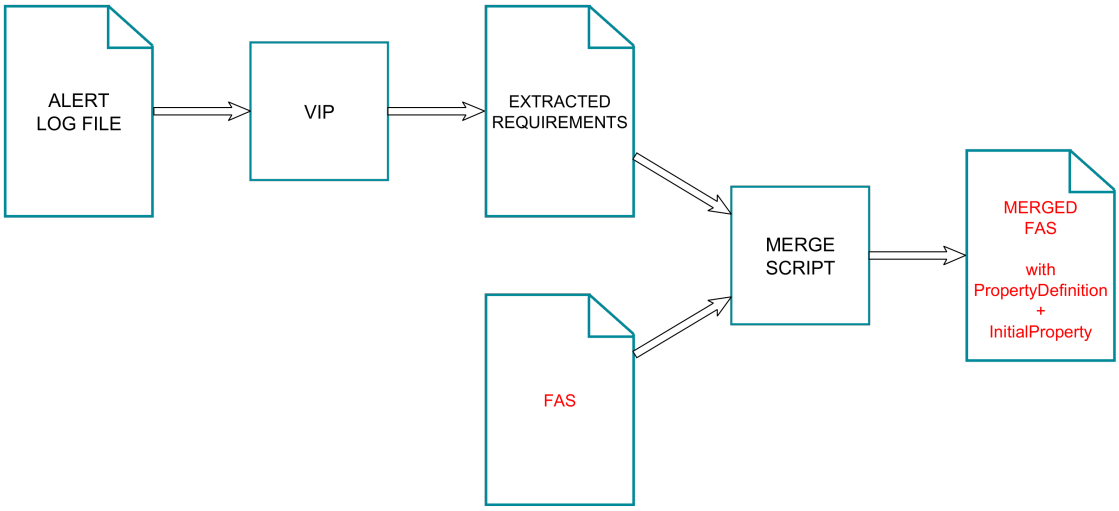

Figure 5.6. Step 4 of 7 (React-VEREFOO)

In the fifth step, the merged FAS is used to generate a new FAS through React-VEREFOO.

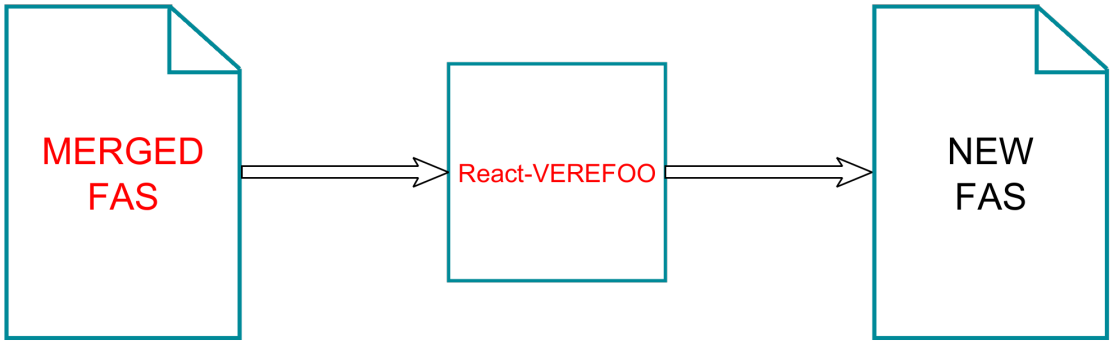

Figure 5.7. Step 5 of 7 (React-VEREFOO)

The sixth step mirrors its counterpart from the previous chapter, with React-VEREFOO replacing VEREFOO for network file generation.

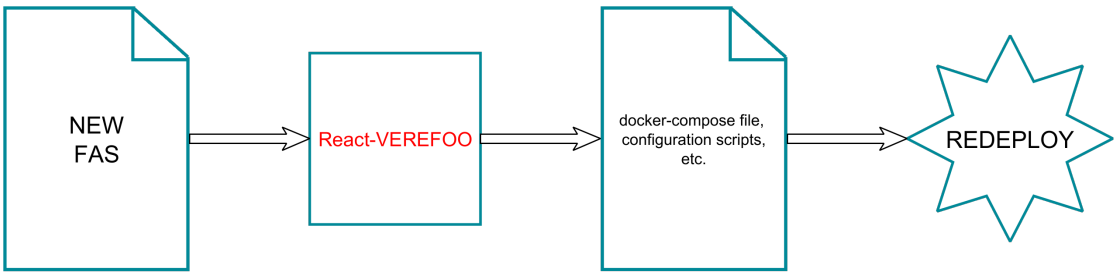

Figure 5.8. Step 6 of 7 (React-VEREFOO)

The final step, involving a repeated attack run to verify the effectiveness of the new requirements, remains as previously described.

The updated approach starts with FAS generation using VEREFOO and transitions to leveraging React-VEREFOO in subsequent steps for an optimized response.

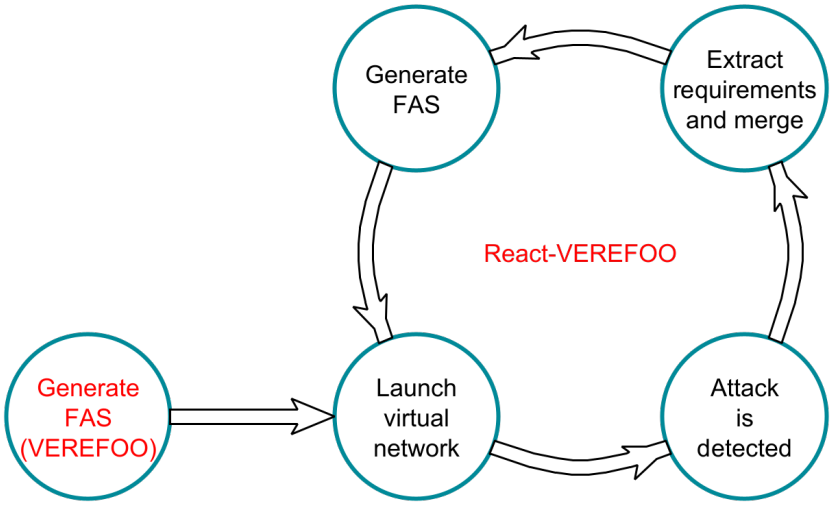

Figure 5.9. Attack response cycle (React-VEREFOO)

Chapter 6

Process automation

6.1 The mergeRequirements algorithm

6.1.1 Introduction

As described in prior chapters, our methodology necessitates integrating newly extracted Network Security Requirements with existing ones. To address this need, the mergeRequirements algorithm was developed. Initially conceived as a standalone Python script, it was subsequently integrated into the vlogi tool to enable automated processing.

The following sections will detail key aspects of the mergeRequirements algorithm. We will explore each stage of the algorithm, starting from the preprocessing steps, delving into the central merging process, and concluding with the postprocessing phase. These sections aim to illustrate the purpose and mechanics of each part of the algorithm.

Finally, we will address the assumptions at the core of the algorithm and examine its limitations in the context of current implementation. This discussion is intended to provide a comprehensive understanding of the algorithm's operational scope and potential areas for future refinement.

6.1.2 Refinement of source and destination IPs

In the initial preprocessing stage of the algorithm, we focus on refining the source and destination IP addresses within the extracted Network Security Requirements. This process aims to align these IP addresses with the most specific corresponding IP or subnet defined in our network topology.

For illustration, consider the network topology depicted in Figure 5.3. Suppose we have extracted the following requirement from OSSEC's alerts:

Listing 6.1. Example requirement extracted from OSSEC's alert file

```
<PropertyDefinition>
    <Property graph="0" name="IsolationProperty" src="192.168.3.1"
    dst="130.10.0.1" lv4proto="TCP"/>
</PropertyDefinition>
```

In this scenario, the IP address `192.168.3.1` would be adjusted to `192.168.3.-1`. This change is necessitated by our network topology, which recognizes the `192.168.3.-1` subnet but not individual endpoint IPs within this range. OSSEC, in its operation, identifies the source IP from system files but lacks contextual knowledge about its subnet association or the broader network topology that VEREFOO processes.

Thus, this preprocessing step systematically replaces the `src` and `dst` IP addresses in each extracted `Property` node with the nearest identifiable IP address or subnet present in the network topology. This refinement ensures that the requirements accurately align with the actual network structure.

Algorithm 1 Refine source and destination IPs to the most specific IP (or subnet) within the topology

Require: *extractedProperties*, *topologyGraph*
for each *eprop* in *extractedProperties* **do**
  for each *position* in {"src", "dst"} **do**
    $ip \leftarrow$ octets from the current IP address
    for $i \leftarrow length(ip) - 1$ to 1 **do**
      if node in *topologyGraph* matching the current subnet **then**
        current IP $\leftarrow$ current subnet
        break
      end if
      $ip[i] \leftarrow$ " $-$ 1"
    end for
  end for
end for

6.1.3 Building NAT IP dictionary

The subsequent preprocessing step involves constructing a dictionary that associates each public NAT IP found within the network topology to its corresponding private IP addresses.

This mapping is essential for the algorithm's processing of requirements that involve public NAT IPs. Specifically, when such a requirement is encountered, the algorithm uses this map to deduce the most probable private network that the public NAT IP is representing. A more detailed explanation of this process will be provided in Subsection 6.1.6.

Algorithm 2 Build a dictionary to associate each NAT's public IP with its corresponding private IP addresses

Require: *topologyGraph*

nats ← all NAT nodes in *topologyGraph*
natIPs ← empty dictionary
for each *nat* in *nats* **do**
    name ← current NAT's public IP
    sources ← all `source` elements within *nat*
    sourceIPs ← empty list
    for each *source* in *sources* **do**
        append IP from *source* to *sourceIPs*
    end for
    natIPs[*name*] ← *sourceIPs*
end for

6.1.4 Building loadbalancer IP dictionary

The third preprocessing step entails creating a dictionary to associate each loadbalancer's IP, as identified in the network topology, with the corresponding IPs in its server pool.

This dictionary is essential for processing requirements involving a loadbalancer IP. When such a requirement is extracted, the algorithm needs to apply the same rule to each server in the loadbalancer's pool. The dictionary facilitates quick generation of these rules for all associated servers. A more detailed explanation of this process will be provided in Subsection 6.1.7.

Algorithm 3 Build a dictionary to associate each loadbalancer's IP with the IPs in its pool

Require: *topologyGraph*

loadbalancers ← all loadbalancer nodes in *topologyGraph*
lbIPs ← empty dictionary
for each *lb* in *loadbalancers* **do**
    name ← current loadbalancer's IP
    pool ← all `pool` elements within *lb*
    poolIPs ← empty list
    for each *poolIP* in *pool* **do**
        append IP from *poolIP* to *poolIPs*
    end for
    lbIPs[*name*] ← *poolIPs*
end for

6.1.5 Setting default values for missing attributes

The purpose of this preprocessing step is to simplify subsequent processing by ensuring consistency across all `Property` nodes. By assigning default values to

any missing attributes, we eliminate discrepancies and achieve uniformity in the attribute set of each `Property`.

Algorithm 4 Set missing attributes to `null` to simplify subsequent checks

procedure AddNullAttributes(*prop*)
    if `lv4proto` attribute not present in *prop* **then**
        add `lv4proto` attribute to *prop* with value `null`
    end if
    if `src_port` attribute not present in *prop* **then**
        add `src_port` attribute to *prop* with value `null`
    end if
    if `dst_port` attribute not present in *prop* **then**
        add `dst_port` attribute to *prop* with value `null`
    end if
end procedure

Require: *extractedProperties*, *topologyProperties*
for each *eprop* in *extractedProperties* **do**
    AddNullAttributes(*eprop*)
end for
for each *tprop* in *topologyProperties* **do**
    AddNullAttributes(*tprop*)
end for

6.1.6 Translating public NAT IPs to private networks

This preprocessing step is dedicated to translating public NAT IPs found within the extracted requirements to the corresponding private network IP(s) that are most likely to be the actual sources.

Consider the network topology depicted in Figure 4.1. In this topology, the traffic monitor, positioned centrally, can only see the public NAT IP in packets coming from or going to private networks behind the NAT. If the traffic monitor detects an attack and generates an alert containing a public NAT IP, this alert, once processed into a VEREFOO requirement by the VIP tool, needs accurate interpretation. Specifically, we need to identify which private network initiated the attack, as blocking all traffic to every private network due to one compromised source is impractical. Therefore, the algorithm assesses the current network requirements to deduce the most likely originating private network(s).

To more fully comprehend, let's analyze the relevant NSRs in our original topology:

Listing 6.2. Relevant NSRs in original network topology

```
<PropertyDefinition>
    <!-- policy 3 -->
    <Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
        dst="130.10.0.4" lv4proto="TCP" />
    <!-- policy 4 -->
```

```
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.4" lv4proto="UDP" />
    <!-- policy 5 -->
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.4" lv4proto="OTHER" />
    <!-- policy 6 -->
    <Property graph="0" name="IsolationProperty" src="192.168.2.-1"
        dst="130.10.0.4" />
    <!-- policy 7 -->
    <Property graph="0" name="ReachabilityProperty" src="130.10.0.4"
        dst="192.168.3.-1" />

    <!-- Repeat policies for all servers within the loadbalancer's pool -->
    <!-- Irrelevant policies have been omitted -->
</PropertyDefinition>
```

In this example, network `192.168.3.-1` can communicate with the loadbalancer at `130.10.0.4` over TCP, and vice versa. However, the `192.168.2.-1` network is prohibited from any communication with the loadbalancer.

Now, consider an extracted requirement indicating an attack:

Listing 6.3. Extracted requirement example involving public NAT IP

```
<PropertyDefinition>
    <Property graph="0" name="IsolationProperty" src="220.124.30.1"
    dst="130.10.0.4" dst_port="7597" lv4proto="TCP"/>
</PropertyDefinition>
```

Upon analysis, the algorithm infers that this requirement most likely refers to the `192.168.3.-1` network. The rationale is that the `192.168.2.-1` network cannot interact with the loadbalancer. Consequently, the algorithm modifies the requirement to:

Listing 6.4. Translated requirement with public NAT IP replaced by private network IP

```
<PropertyDefinition>
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
    dst="130.10.0.4" dst_port="7597" lv4proto="TCP"/>
</PropertyDefinition>
```

Had there been multiple matching private networks, the algorithm would have replicated the isolation rule for each applicable network.

Algorithm 5 Translate public NAT IPs to corresponding private networks

Require: *extractedProperties*, *topologyProperties*, *natIPs*

```
1: natProperties ← empty list
2: for each eprop in extractedProperties do
3:     srcNat ← true if current src IP is a public NAT IP in natIPs
4:     dstNat ← true if current dst IP is a public NAT IP in natIPs
5:     if srcNat and not dstNat then
6:         privateSrcs ← private IP(s) deduced to be the actual source(s)
7:         for each privateSrc in privateSrcs do
8:             create a new property based on eprop but with privateSrc as src IP
9:             append the new property to natProperties
10:        end for
11:    else if dstNat and not srcNat then
12:        privateDsts ← private IP(s) deduced to be the actual destination(s)
13:        for each privateDst in privateDsts do
14:            create a new property based on eprop but with privateDst as dst IP
15:            append the new property to natProperties
16:        end for
17:    else if srcNat and dstNnat then
18:        for each combination of privateSrc and privateDst do
19:            for each tprop in topologyProperties do
20:                if current property matches with tprop then
21:                    create a new property based on eprop but with privateSrc as
                       source IP and privateDst as destination IP
22:                    append the new property to natProperties
23:                end if
24:            end for
25:        end for
26:    end if
27: end for
28: Remove elements with public NAT IPs from extractedProperties
29: Append natProperties to extractedProperties
```

6.1.7 Expanding properties for loadbalancers

The last preprocessing step before merging the requirements involves handling `Property` elements that reference a loadbalancer IP. In this step, for each `Property` associated with a loadbalancer IP, new properties are created using the IP addresses from the loadbalancer's pool, while preserving the original `Property`.

This replication is essential to ensure that the set of requirements fed into VEREFOO is complete. Without duplicating the `Property` for each server in the pool, there could be gaps in the requirements, potentially leading to issues in the generation of the FAS.

Algorithm 6 For each `Property` referencing a loadbalancer, create new properties using IPs from its pool, retaining the original `Property`

Require: $extractedProperties$, $lbIPs$

$lbProperties \leftarrow$ empty list
for each *eprop* in $extractedProperties$ **do**
    $srcLb \leftarrow$ true if current src IP is a loadbalancer IP in $lbIPs$
    $dstLb \leftarrow$ true if current dst IP is a loadbalancer IP in $lbIPs$
    if $srcLb$ and not $dstLb$ **then**
        for each $lbSrcIP$ matching the current src loadbalancer IP in $lbIPs$ **do**
            create a new property based on *eprop* but with $lbSrcIP$ as src IP
            append the new property to $lbProperties$
        end for
    else if $dstLb$ and not $srcLb$ **then**
        for each $lbDstIP$ matching the current dst loadbalancer IP in $lbIPs$ **do**
            create a new property based on *eprop* but with $lbDstIP$ as dst IP
            append the new property to $lbProperties$
        end for
    else if $srcLb$ and $dstLb$ **then**
        for each combination of $lbSrcIP$ and $lbDstIP$ **do**
            create a new property based on *eprop* but with $lbSrcIP$ as src IP and $lbDstIP$ as dst IP
            append the new property to $lbProperties$
        end for
    end if
end for
Append $lbProperties$ to $extractedProperties$

6.1.8 Merging NSRs

The merging of NSRs constitutes the core functionality of the algorithm. Following all preprocessing steps, the algorithm can now safely merge current requirements with newly extracted ones.

The merging process involves two distinct scenarios:

1. **Exact match:** If an extracted isolation requirement exactly matches a current reachability requirement, the reachability requirement is removed from the final list, and the isolation requirement is added.

2. **Loose match:** If an extracted isolation requirement loosely matches a current reachability requirement (i.e., the isolation requirement is a subset of the reachability requirement), the reachability requirement is replaced by other reachability requirements that allow the same traffic as the original, except for the portion blocked by the isolation requirement. Subsequently, the isolation requirement itself is added to the final list.

The `DeriveReachabilityProperties` function is responsible for generating these new reachability requirements. To illustrate its functionality, consider the

NSRs presented in Listing 6.2. Suppose a new requirement is extracted, as shown in Listing 6.3.

After the necessary preprocessing steps, we obtain the output depicted in Listing 6.4. This output indicates a policy that blocks all TCP traffic on port 7597 from `192.168.3.-1` to `130.10.0.4`. Since the original NSRs permitted all TCP traffic, modifications are required to align with this new constraint. To achieve this, the original NSRs are adapted as follows:

Listing 6.5. Merged requirements

```
<PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
    dst="130.10.0.4" lv4proto="TCP" dst_port="0-7596" />
    <Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
        dst="130.10.0.4" lv4proto="TCP" dst_port="7598-65535" />
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.4" dst_port="7597" lv4proto="TCP" />

    <!-- Repeat policies for all servers within the loadbalancer's pool -->
    <!-- Irrelevant policies have been omitted -->
</PropertyDefinition>
```

In this adaptation, the original NSR permitting all TCP traffic is split into two, allowing traffic on ports 0-7596 and 7598-65535, effectively isolating only port 7597 as dictated by the new requirement.

Algorithm 7 Merge properties

Require: *extractedProperties*, *topologyProperties*

mergedPropertiesList ← copy of *topologyProperties*
for each *eprop* in *extractedProperties* **do**
    index ← 0
    while *index* < *length*(*mergedPropertiesList*) **do**
        mprop ← *mergedPropertiesList*[*index*]
        if *eprop* exactly matches with *mprop* **then**
            remove *mprop* from *mergedPropertiesList*
            break
        end if
        if *eprop* loosely matches with *mprop* **then**
            remove *mprop* from *mergedPropertiesList*
            toAppend ← DeriveReachabilityProperties(*mprop*, *eprop*)
            for each *ta* in *toAppend* **do**
                append *ta* to *mergedPropertiesList* if not duplicate
            end for
            break
        end if
        index ← *index* + 1
    end while
    append *eprop* to *mergedPropertiesList* if not duplicate
end for

Algorithm 8 Derive reachability properties based on isolation constraints

```
1: function DeriveReachabilityProperties(mprop, eprop)
2:     mproto ← lv4proto from mprop
3:     eproto ← lv4proto from eprop
4:     msrcPort ← src_port from mprop
5:     esrcPort ← src_port from eprop
6:     mdstPort ← dst_port from mprop
7:     edstPort ← dst_port from eprop
8:     properties ← empty list
9:
10:    if eproto is TCP and mproto is null then
11:        for each p in [UDP, OTHER] do
12:            append reachability property with protocol p to properties
13:        end for
14:    else if eproto is UDP and mproto is null then
15:        for each p in [TCP, OTHER] do
16:            append reachability property with protocol p to properties
17:        end for
18:    else if eproto is OTHER and mproto is null then
19:        for each p in [TCP, UDP] do
20:            append reachability property with protocol p to properties
21:        end for
22:    end if
23:
24:    srcPortElements ← empty list
25:    dstPortElements ← empty list
26:
27:    if esrcPort is not null then
28:        generate reachability properties for src port ranges not blocked by eprop
29:        add them to srcPortElements
30:    end if
31:    if edstPort is not null then
32:        generate reachability properties for dst port ranges not blocked by eprop
33:        add them to dstPortElements
34:    end if
35:
36:    if no srcPortElements then
37:        append dstPortElements to properties
38:    else if no dstPortElements then
39:        append srcPortElements to properties
40:    else
41:        combine srcPortElements with dstPortElements and append to
           properties
42:    end if
43:
44:    return properties
45: end function
```

6.1.9 Postprocessing

In the postprocessing phase of the algorithm, the initial task is to remove any `null` attributes from the merged properties. These attributes, which were useful during previous processing stages, are now redundant and can be safely discarded.

Following this cleanup, the next step involves preparing the data for output. This is done by encapsulating the merged properties within a `PropertyDefinition` element. Simultaneously, the original properties that were initially within the `PropertyDefinition` element are relocated to a new `InitialProperty` element. This enables React-VEREFOO to effectively analyze and compare the old and new sets of network requirements to make decisions about the necessary network reconfigurations.

6.1.10 Assumptions made by the algorithm

The initial set of requirements must be free of conflicts and suboptimal configurations.

Each private network behind a NAT is unique within the network topology. As detailed in Subsection 6.1.6, the algorithm translates public NAT IPs in the extracted requirements to the most probable corresponding private network IP(s). If two different networks behind different NATs share identical IPs, this could lead to ambiguities in the Firewall Allocation Scheme generation.

The algorithm assumes there are no traffic monitors between the server pool and the loadbalancer. Extracted requirements are presumed to reference the loadbalancer's IP, not the IPs of individual servers in the pool. This assumption ensures uniform treatment of servers within a pool, as explained in Subsection 6.1.7.

Initial requirements involving NATs should use private IP addresses, aligning with the algorithm's approach of translating public NAT IPs to private addresses.

Servers within a loadbalancer's pool must be clearly defined as network nodes to facilitate the duplication of requirements for each server in the pool.

Loadbalancers are not positioned behind NATs. The algorithm does not cover this scenario because a loadbalancer itself functions similarly to a NAT, and accommodating such a configuration would significantly increase the algorithm's complexity.

As mentioned in Subsection 6.1.8, the algorithm operates under the assumption that isolation requirements either exactly match or are subsets of current reachability requirements. For instance, a reachability requirement that permits TCP traffic from a specific source to a destination cannot be merged with an isolation requirement that denies all traffic from that source to the destination. This assumption stems from the fact that if specific traffic is explicitly allowed, then detected attacks are expected to be related to that specific traffic or a subset thereof, not to traffic that was never permitted initially.

6.1.11 Limitations

The algorithm, while operational, is still in its early stages of development and requires more comprehensive testing to ensure reliability in various scenarios.

At this stage, although it includes small optimizations and preprocessing steps to enhance efficiency, there is ample scope for advancement. Enhancements are needed to ensure the script can handle more complex cases efficiently and effectively, which is essential for its broader applicability and robustness in diverse network settings.

6.2 VEREFOO Log Integrator

The VEREFOO Log Integrator (vlogi) is a command-line utility written in Python and specifically developed to monitor logs from Intrusion Detection Systems in a VEREFOO-based network environment. Its primary function is to detect new log entries, which indicate potential security incidents. Once such an entry is identified, vlogi automatically executes all necessary actions to enable a prompt and efficient response to active attacks.

vlogi is designed to be compatible with alert modes and formats used by the VIP tool. It supports processing of log files in both plaintext and JSON formats. This includes the capability to handle logs generated using the `AlertFastV0` strategy, which outputs in plaintext, as well as those from the `JsonOut` strategy, where logs are formatted in JSON.

A key feature of vlogi is its extensibility. The tool is structured to allow users to easily add new IDS and alert mode combinations for monitoring. This is facilitated through variables set in its configuration file. Moreover, vlogi is structured to potentially interface with external programs, offering the flexibility to filter logs before integrating them into the network's security measures.

The primary objective of this section is to present a comprehensive overview of vlogi's architecture. It explains the roles and functionalities of each component within vlogi, illustrates the interaction between these components, and provides guidance on configuring and using the software, as well as on future enhancements and expansions.

6.2.1 Motivation and need

The development of vlogi was driven by the need to automate the attack response process within the VEREFOO framework. This automation represents a practical implementation of the process described in Subsection 5.3.4, streamlining the response mechanism to potential security threats in the network.

6.2.2 Architectural overview

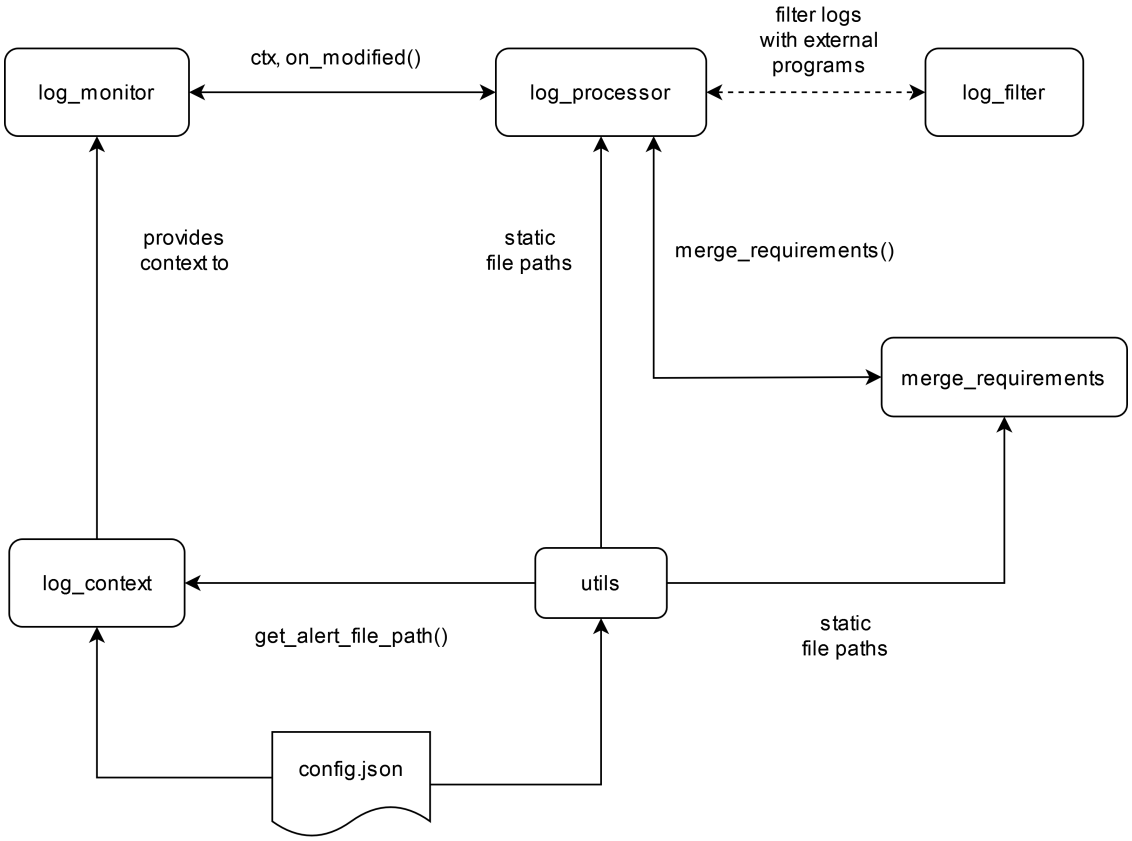

Figure 6.1. Diagram of vlogi's main components and their interactions

The above diagram provides a visual representation of the vlogi tool, outlining the interconnections between its components and their respective roles.

Upon startup, vlogi retrieves command-line arguments alongside configuration variables from the `config.json` file. Using the information from these two sources, it determines the path to the alert file for the current session. Once this path is established, vlogi creates an object called `LogContext`, which stores the combined information from `config.json`, the command-line arguments, and the newly determined alert file path.

Following this, vlogi instantiates a `LogMonitor`, providing it with the newly created `LogContext` object. With the `LogMonitor` in place, vlogi is then ready to begin the log monitoring process, which is initiated by invoking the `start()` method of the `LogMonitor`.

Once a new alert is generated, the `LogProcessor` is activated through its `on_modified()` method. This activation captures the log entries, which can be further refined by the `LogFilter`, updating the NSRs as a result. The subsequent steps taken by vlogi depend on the `auto_confirm` setting; it may either autonomously apply the updates or request user verification. Following approval, vlogi generates new configuration files, enforces the updates, and resumes log monitoring.

The `merge_requirements` module, housing the algorithm described in Section 6.1, is tasked with the critical function of merging the existing NSRs with those extracted from log entries, ensuring that the network's security policies are fortified against ongoing threats.

Additionally, the `utils` module includes the `get_alert_file_path()` function, which, as previously mentioned, determines the alert log file path during the startup phase. It also contains predefined constants and static paths to VEREFOO and virtual network files, which are used by other modules within vlogi.

6.2.3 Configuration and usage

Prior to running vlogi, it is necessary to follow specific steps to ensure the tool operates correctly:

- Confirm that the `firewall-type` variable in the `config.json` file is correctly configured to reflect the particular firewall software to deploy.
- Adjust the current working directory to the root project folder, not the inner `vlogi` folder. This is crucial to avoid any potential path errors.
- Launch VEREFOO and VIP, then generate the initial Firewall Allocation Scheme along with the set of virtual network files.
- In the root project folder, create a subdirectory named `verefoo_network_files`. Place the topology file and the FAS generated in the previous step into this subdirectory.
- Relocate the `vnetwork` folder, which contains the previously generated virtual network files, to the root project folder.

To execute vlogi, follow these steps:

- Launch the tool using the command `python3 vlogi` and include the name of the IDS to be monitored, its version, and the format in which the IDS reports alerts.
- Depending on the specific IDS in use, it might be necessary to specify the type of installation. This is done using the `-i` option, where the installation type can be specified as `server`, `agent`, `local`, or `hybrid`.
- Use the `-a` flag to enable the `auto_confirm` feature, allowing vlogi to integrate new requirements without requiring manual user confirmation.
- Set the minimum priority level required for alert processing using the `-p` option.

6.2.4 Extending vlogi

vlogi offers four key areas for potential extensions:

Adding new modules

To incorporate a new module into vlogi:

- Create a new directory within the inner `vlogi` folder and name it after the module.
- Develop the core logic of your module. You can distribute this logic over several Python files if necessary.
- Include an `__init__.py` file in your module's directory. This file determines which components of your module are accessible to other parts of the application.

Adding new IDS/alert mode combinations

To integrate new IDS/alert mode combinations:

- Ensure that the `ids`, `version`, and `alert_mode` parameters in vlogi align with those in the VIP tool.
- After incorporating a new IDS/alert mode into the VIP tool, the `config.json` file in vlogi needs to be updated. This update involves adding the new combination under the `logfiles` key.

Supporting additional firewall types

In regards to expanding firewall support:

- The necessary code to support additional firewall types has already been included.
- Once support for a new firewall type is added to VEREFOO's virtual network translator, enable its integration with vlogi by uncommenting the relevant sections in the `utils/firewall_types.py` file.

Implementing the `LogFilter` class

Concerning the development of the `LogFilter` class:

- Currently, it includes two empty static methods: `script_filter()` and `api_filter()`.
- These methods are intended for preliminary filtering of logs before they undergo full processing.

6.2.5 Putting it all together

Building upon the foundation laid out in Subsection 5.3.4, we have successfully automated the response process in vlogi. The initial steps remain manual, but as we transition into the cyclic phase of the process illustrated in Figure 5.9, vlogi takes the lead. It automatically responds to ongoing attacks by extracting new requirements from alerts, merging these with the manually generated FAS, and then producing and deploying the updated virtual network files.

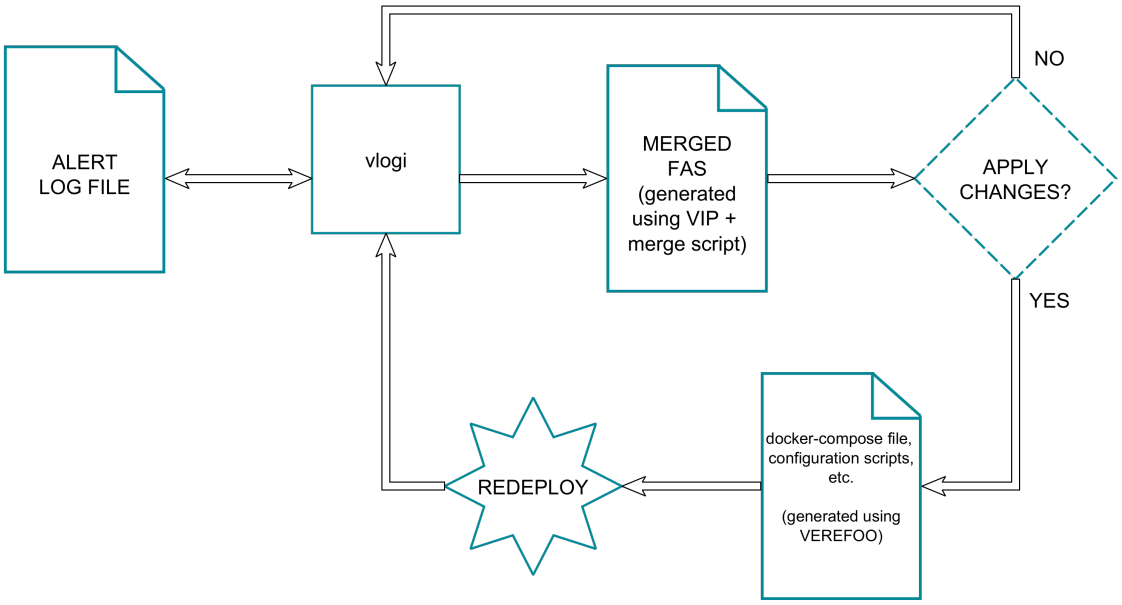

Figure 6.2. Automated response process within vlogi

In essence, the workflow mirrors that presented in Figure 5.9, with the notable distinction being the complete automation of the cyclic portion:

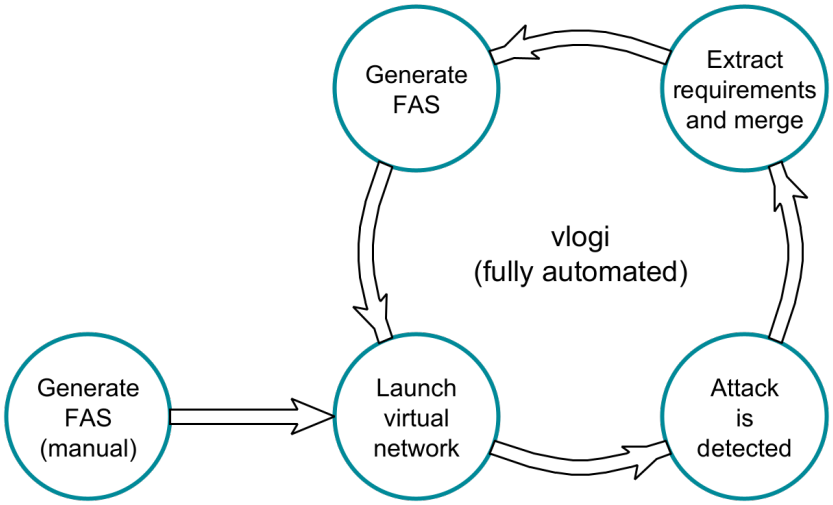

Figure 6.3. Attack response cycle (vlogi)

Chapter 7

Demo validation

This chapter provides a detailed hands-on demonstration. It starts with a VEREFOO topology file that depicts the network as shown in Figure 5.3. The process involves generating a Firewall Allocation Scheme, using it to create a virtual network and its corresponding configuration files, and then deploying the network. Subsequently, the process includes starting vlogi, executing an attack, and observing how vlogi autonomously blocks the attack by reconfiguring the network.

7.1 Prerequisites

The network depicted in Figure 5.3 serves as the foundation for this demo. The security requirements for this network, as defined in the initial topology file, are as follows:

Listing 7.1. NSRs for network shown in Figure 5.3

```
<PropertyDefinition>
    <!-- policy 1 -->
    <Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
        dst="130.10.0.1" lv4proto="TCP" />
    <Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
        dst="130.10.0.2" lv4proto="TCP" />
    <Property graph="0" name="ReachabilityProperty" src="192.168.3.-1"
        dst="130.10.0.3" lv4proto="TCP" />
    <!-- policy 2 -->
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.1" lv4proto="UDP" />
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.2" lv4proto="UDP" />
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.3" lv4proto="UDP" />
    <!-- policy 3 -->
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.1" lv4proto="OTHER" />
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.2" lv4proto="OTHER" />
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.3" lv4proto="OTHER" />
    <!-- policy 4 -->
```

```
    <Property graph="0" name="ReachabilityProperty" src="130.10.0.1"
        dst="192.168.3.-1" lv4proto="ANY" />
    <Property graph="0" name="ReachabilityProperty" src="130.10.0.2"
        dst="192.168.3.-1" lv4proto="ANY" />
    <Property graph="0" name="ReachabilityProperty" src="130.10.0.3"
        dst="192.168.3.-1" lv4proto="ANY" />
</PropertyDefinition>
```

The network consists of a straightforward setup: it includes three distinct servers linked to an endpoint network through a forwarder, and one of these servers (`server1`) is equipped with OSSEC 3.7. The security requirements are minimal, allowing all TCP traffic between the endpoint network and the servers.

For the generation of the Firewall Allocation Scheme and to enable vlogi's automated response to attacks, VEREFOO, React-VEREFOO, and VIP are required. The initial FAS is produced using standard VEREFOO, while subsequent versions are dynamically generated by vlogi employing React-VEREFOO and VIP. A critical aspect of this setup is running VEREFOO and React-VEREFOO on distinct ports. This is necessary because they are both configured by default to use port 8085, which would lead to a conflict.

For the purpose of this demonstration, it is presumed that the working directory is set to vlogi's root project folder, in line with the guidelines provided in Subsection 6.2.3. Once the necessary `.jar` files are prepared, the following commands are used to start VEREFOO on port 8086, React-VEREFOO on port 8085, and VIP on port 8080:

Listing 7.2. Launch VEREFOO React-VEREFOO and VIP

```
java -jar path/to/verefoo.jar --server.port=8086
java -jar path/to/react-verefoo.jar
java -jar path/to/vip.jar
```

7.2 Generating the Firewall Allocation Scheme

With all preparations complete, the first step is to create the initial FAS. This is achieved using the command below:

Listing 7.3. Generate the initial FAS

```
curl -X POST http://localhost:8086/verefoo/adp/simulations -H
    "accept:␣application/xml" -H "Content-Type:␣application/xml"
    -d @path/to/Topology.xml > path/to/FAS.xml
```

The generated FAS is identical to the original topology file, except for one difference: the introduction of a firewall between the forwarder and the endpoint network. In adherence to the NSRs specified in the topology file, this firewall is configured to allow all TCP traffic while concurrently blocking everything else:

Listing 7.4. Firewall node within the initial FAS

```
<!-- [...] -->
```

```
<node name="1.0.0.4" functional_type="FIREWALL">
    <neighbour name="33.33.33.2"/>
    <neighbour name="192.168.3.-1"/>
    <configuration name="AutoConf" description="1">
        <firewall defaultAction="DENY">
            <elements>
                <action>ALLOW</action>
                <source>-1.-1.-1.-1</source>
                <destination>-1.-1.-1.-1</destination>
                <protocol>TCP</protocol>
                <src_port>*</src_port>
                <dst_port>*</dst_port>
            </elements>
        </firewall>
    </configuration>
</node>
<!-- [...] -->
```

Below is a visual representation of the FAS, showcasing the integration of the firewall:

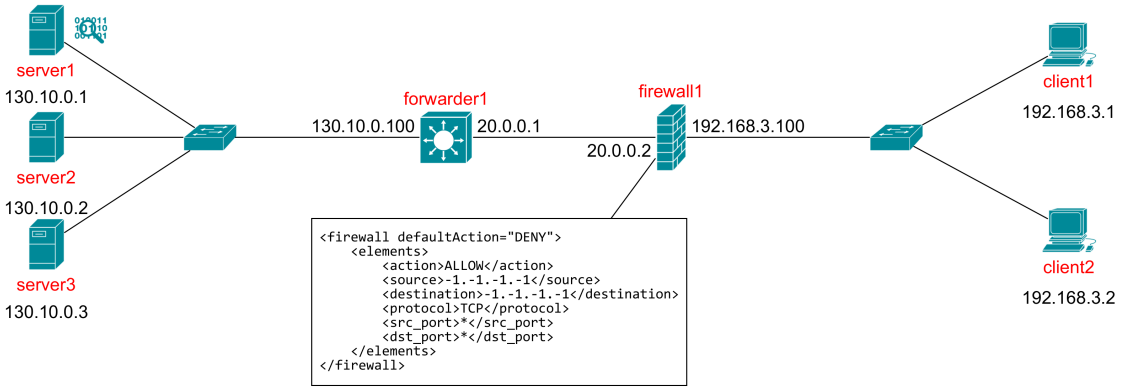

Figure 7.1. Visual representation of the initial FAS

Prior to advancing to the next step of the process, and as indicated in Subsection 6.2.3, a directory named `verefoo_network_files` needs to be created inside vlogi's root project folder. The purpose is to store the topology file and the FAS. The following command illustrates how to create this directory and move the files:

Listing 7.5. Move topology and FAS files into `vlogi/verefoo_network_files`

```
mkdir verefoo_network_files
mv path/to/Topology.xml path/to/FAS.xml verefoo_network_files
```

7.3 Launching the virtual network

With the FAS in place, the next step is to generate the required files for the virtual network and proceed with its launch. This demonstration involves deploying firewalls configured as iptables. Consequently, this choice is specified in the following

request to ensure the correct firewall type is deployed:

Listing 7.6. Generate virtual network files

```
curl -X POST \
"http://localhost:8085/verefoo/adp/venvironment/"\
"generateFiles?firewallType=IPTABLES" \
-H "Content-Type:␣application/xml" \
-d @verefoo_network_files/FAS.xml
```

Executing this command results in the creation of a folder named `vnetwork` inside vlogi's root project folder, which houses all the essential files needed to launch the network.

The final step before launching the network is to generate the firewall configuration files. In this network configuration, only one firewall is present, requiring the generation of a single configuration file. If the network included multiple firewalls, a corresponding number of configuration files would need to be generated.

Before generating the configuration file, it is necessary to store the FAS within React-VEREFOO:

Listing 7.7. Store FAS inside React-VEREFOO

```
curl -X POST http://localhost:8085/verefoo/fwd/nodes/addnfv -H
   "accept:␣application/xml" -H "Content-Type:␣application/xml"
   -d @verefoo_network_files/FAS.xml
```

Following this, the firewall configuration file is generated using the command below:

Listing 7.8. Generate the firewall configuration file

```
curl -X GET \
http://localhost:8085/verefoo/fwd/deploy/getIptables/1 \
-H "accept:␣*/*" \
-o vnetwork/FirewallConfig/iptables/iptablesFirewall_1_1.sh
```

Prior to launching the virtual network, it is important to ensure that all script files in the `vnetwork` folder are granted execute permissions:

Listing 7.9. Grant execute permissions to all script files inside `vnetwork`

```
find vnetwork -type f -name "*.sh" -exec chmod +x {} \;
```

With these preparations completed, the virtual network is ready to be activated. It is important to specify the deployment of iptables firewalls when executing the start script:

Listing 7.10. Start the virtual network

```
cd vnetwork && sudo ./startScript.sh iptables
```

7.4 Running vlogi

The final manual action required is starting vlogi. Since iptables firewalls have been deployed throughout the network, ensuring that the `firewall-type` key in the `config.json` file is set to `IPTABLES` is essential. This setting informs vlogi about the specific type of firewall in use, enabling it to generate the correct configuration files.

With OSSEC 3.7 installed locally on `server1`, the command to start vlogi is as follows:

Listing 7.11. Start vlogi

```
python3 vlogi ossec 3.7 jsonout -i local
```

Once started, vlogi will begin monitoring OSSEC's alert file to detect and respond to new security attacks.

7.5 Attack simulation

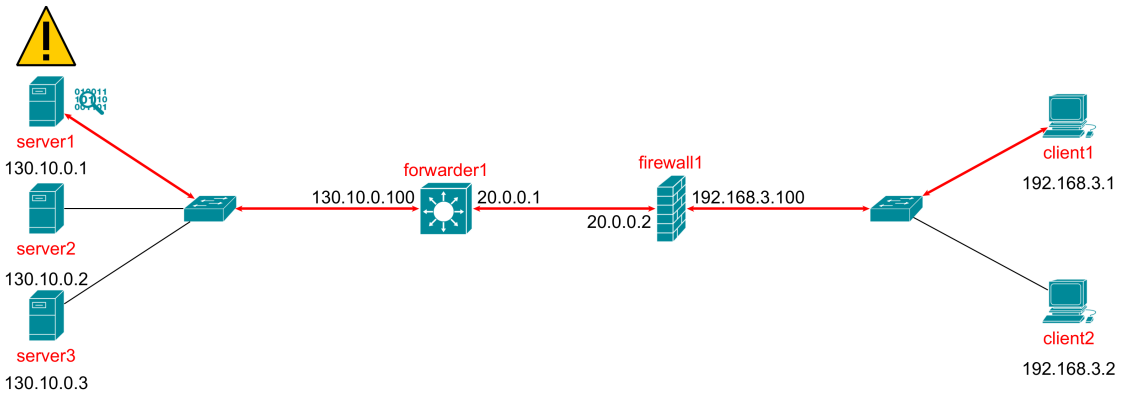

Figure 7.2. Port scan attack simulation

The next step is to simulate an attack, specifically a port scan, targeting `server1` from `client1`. Port scans are commonly encountered in cyber attacks and are relatively straightforward to detect. Being a host-based IDS, OSSEC typically does not detect network-based attacks directly. To address this, `server1` has been set up with iptables to create logs, while OSSEC has been configured to analyze these logs in order to identify port scans. The rules added to OSSEC's `local_rules.xml` for detecting port scans are as follows:

Listing 7.12. Rules added to `local_rules.xml` for detecting port scans

```
<rule id="100009" level="1">
    <options>no_log</options>
    <decoded_as>iptables</decoded_as>
    <description>Individual TCP SYN request detected</description>
```

```
</rule>

<rule id="100010" level="10" frequency="20" timeframe="60">
    <if_matched_sid>100009</if_matched_sid>
    <decoded_as>iptables</decoded_as>
    <description>Possible TCP SYN port scan detected</description>
    <same_source_ip />
</rule>
```

Rule `100009` is designed to match any TCP SYN request without logging it. This rule assists rule `100010`, which activates when at least 20 TCP SYN requests from the same source IP are identified within 60 seconds, signaling a potential port scan.

The attack simulation starts by accessing `client1` via a shell:

Listing 7.13. Open a shell to `client1`

```
sudo docker exec -it client1 /bin/sh
```

The port scan is simulated through a simple command employing `netcat`. This method is preferred over just using `nmap` to minimize the need for extra software installations on endpoints. The specific command used for the attack simulation is:

Listing 7.14. Simulate a port scan

```
for port in $(seq 1 25); do nc 130.10.0.1 $port; done
```

This command sends 25 TCP SYN requests to `server1`, triggering the rule in OSSEC and resulting in the following alert:

Listing 7.15. Alert generated by OSSEC

```
{
  "rule": {
    "level": 10,
    "comment": "Possible TCP SYN port scan detected",
    "sidid": 100010,
    "frequency": 20,
    // [...]
  },
  // [...]
  "protocol": "TCP",
  "srcip": "192.168.3.1",
  "dstip": "130.10.0.1",
  // [...]
  "agent_name": "server1",
  "timestamp": "2023 Dec 13 10:35:57",
  "logfile": "/var/log/ulog/syslogemu.log"
}
```

vlogi then processes this alert, converting it into a VEREFOO NSR using the VIP tool:

Listing 7.16. OSSEC alert converted to VEREFOO NSR

```
<PropertyDefinition>
    <Property graph="0" name="IsolationProperty" src="192.168.3.1"
    dst="130.10.0.1" lv4proto="TCP"/>
</PropertyDefinition>
```

Subsequently, vlogi integrates this requirement with existing ones in the initial FAS using the `merge_requirements` module. This process results in two distinct sets of NSRs: the merged NSRs, now located within the `PropertyDefinition` node, and the original NSRs, retained within the `InitialProperty` node:

Listing 7.17. Merged network requirements

```
<PropertyDefinition>
    <Property name="ReachabilityProperty" graph="0" src="192.168.3.-1"
    dst="130.10.0.2" lv4proto="TCP" isSat="true" />
    <Property name="ReachabilityProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.3" lv4proto="TCP" isSat="true" />
    <Property name="IsolationProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.1" lv4proto="UDP" isSat="true" />
    <Property name="IsolationProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.2" lv4proto="UDP" isSat="true" />
    <Property name="IsolationProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.3" lv4proto="UDP" isSat="true" />
    <Property name="IsolationProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.1" lv4proto="OTHER" isSat="true" />
    <Property name="IsolationProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.2" lv4proto="OTHER" isSat="true" />
    <Property name="IsolationProperty" graph="0" src="192.168.3.-1"
        dst="130.10.0.3" lv4proto="OTHER" isSat="true" />
    <Property name="ReachabilityProperty" graph="0" src="130.10.0.1"
        dst="192.168.3.-1" lv4proto="ANY" isSat="true" />
    <Property name="ReachabilityProperty" graph="0" src="130.10.0.2"
        dst="192.168.3.-1" lv4proto="ANY" isSat="true" />
    <Property name="ReachabilityProperty" graph="0" src="130.10.0.3"
        dst="192.168.3.-1" lv4proto="ANY" isSat="true" />
    <Property graph="0" name="IsolationProperty" src="192.168.3.-1"
        dst="130.10.0.1" lv4proto="TCP" />
</PropertyDefinition>
<InitialProperty>
    <!-- Same policies as before -->
</InitialProperty>
```

At this point, vlogi informs the user that it has detected an attack and reconfigured the network. It then asks the user if they wish to apply the changes. This prompt occurs because the `auto_confirm` flag was not set in the initial launch parameters of vlogi. Had this flag been enabled, vlogi would have automatically applied the changes without requiring confirmation. Upon user approval, vlogi produces the updated FAS. In this revised version, `firewall1` is configured to allow TCP traffic exclusively between `server2`, `server3`, and the endpoint network, effectively isolating the endpoint network from `server1`:

Listing 7.18. New firewall rules

```
<node name="1.0.0.4" functional_type="FIREWALL">
    <neighbour name="33.33.33.2"/>
```

```
        <neighbour name="192.168.3.-1"/>
        <configuration name="AutoConf" description="1">
            <firewall defaultAction="DENY">
                <elements>
                    <action>ALLOW</action>
                    <source>130.10.0.-1</source>
                    <destination>192.168.3.-1</destination>
                    <protocol>ANY</protocol>
                    <src_port>*</src_port>
                    <dst_port>*</dst_port>
                </elements>
                <elements>
                    <action>ALLOW</action>
                    <source>192.168.3.-1</source>
                    <destination>130.10.0.3</destination>
                    <protocol>TCP</protocol>
                    <src_port>*</src_port>
                    <dst_port>*</dst_port>
                </elements>
                <elements>
                    <action>ALLOW</action>
                    <source>192.168.3.-1</source>
                    <destination>130.10.0.2</destination>
                    <protocol>TCP</protocol>
                    <src_port>*</src_port>
                    <dst_port>*</dst_port>
                </elements>
            </firewall>
        </configuration>
    </node>
```

7.6 Verifying the results

Following the automatic reconfiguration of the network by vlogi, the final step is to confirm that the new setup effectively blocks a port scan from `client1` to `server1`. This verification can be accomplished by executing the command presented in Listing 7.14 again.

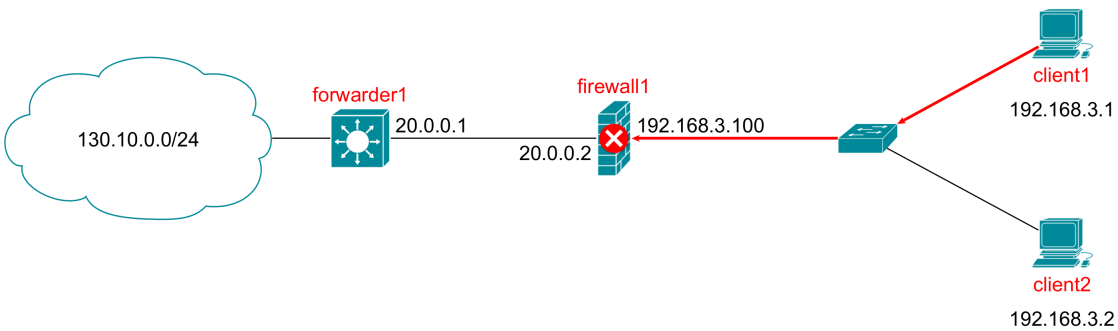

Figure 7.3. Port scan is blocked by the firewall

Chapter 8

Conclusions and future work

8.1 Key achievements

This thesis focused on leveraging IDS alerts for automatically reconfiguring networks using the VEREFOO framework. The central challenge was to develop a system, complete with all necessary components, capable of accomplishing this task. A foundational step involved creating the VIP tool, designed to translate various alerts from different IDSs into VEREFOO NSRs. This tool was developed with a focus on modularity and ease of extensibility, simplifying the addition of support for new alert modes and IDSs. Extensive testing has verified the reliability of the VIP tool at this stage of research.

A pivotal aspect of this research was establishing a cyclic procedure to enable continuous defense against cyber attacks. This approach demonstrated the feasibility of an entirely automated system that can autonomously protect a network from cyber threats.

Another significant part of this research was developing the mergeRequirements algorithm. While still in its early stages, this algorithm plays a crucial role in merging the existing requirements with those derived from IDS alerts. Given the intricate nature of this task, the algorithm is designed to be both efficient and comprehensive. While not yet optimal, the algorithm compensates by incorporating various preprocessing steps and early exit strategies to enhance its efficiency. A noteworthy feature of the algorithm is its ability to determine the corresponding private networks for public NAT IP addresses inside extracted requirements. In addition to that, the algorithm is also designed to address complex scenarios, such as environments with NATs and loadbalancers, ensuring appropriate handling of these network configurations.

The integration of the virtual network translator into VEREFOO was crucial for actual network deployment or reconfiguration based on the initial and extracted requirements. Introducing React-VEREFOO into the process aimed to accelerate the reconfiguration procedure. However, React-VEREFOO's initial tendency to reconfigure every `FORWARDER` node, regardless of its origin, posed a challenge. This issue was resolved by introducing a new VFORWARDER functional type, as detailed in Subsection 5.3.2.

The integration of all components into the vlogi tool marked a significant step towards complete defense automation. vlogi automates the process, requiring manual intervention only for the initial step of building the network topology file and generating initial configurations. The subsequent processes are managed autonomously by vlogi.

Additionally, a demonstration script was developed to detail the complete process. This script serves as a step-by-step guide for users, enhancing comprehension and providing valuable insights for future project maintainers. It features two test networks: the first employs Snort 3 as a network-based IDS, while the second utilizes OSSEC 3.7, installed locally on a server, as a host-based IDS. This setup offers a diverse and practical learning experience, showcasing the system's adaptability to different IDS configurations.

Alongside this, minor bugs in the virtual network translator were addressed, and small enhancements were made, boosting its functionality and reliability.

8.2 Limitations of the current approach

Some limitations in the current approach lie within the foundational assumptions of the mergeRequirements algorithm, as elaborated in Subsection 6.1.10. Being in its nascent stage, the algorithm necessitates additional refinement, particularly in terms of efficiency. Although it has undergone initial testing, more comprehensive and diverse network configuration tests are necessary for its validation and optimization.

Another limitation is vlogi's assumption of only one IDS within the network topology. While deploying multiple instances of vlogi is possible for networks with more than one IDS, the lack of a synchronization mechanism among these instances poses a challenge. Thus, it is currently advisable to operate with a single IDS and one instance of vlogi per network.

The approach currently supports only Snort 3 and OSSEC 3.7 (local installation).

The virtual network translator, in its current state, lacks the generalization and optimization necessary for use in production environments.

8.3 Next steps

Future work by maintainers should aim at enhancing the efficiency of the mergeRequirements algorithm. It is crucial to test it in a variety of network configurations to ensure its adaptability and effectiveness.

Other enhancements should include extending vlogi to support multiple different IDSs in one network. Multi-agent IDS setups can be achieved either natively, by configuring multiple OSSEC `agent` installations to send data to a `server` installation, or alternatively, by integrating OSSEC with other IDSs such as Snort. In this

latter approach, OSSEC would be configured to receive and act upon events generated by Snort. Another important improvement for vlogi is the implementation of a fully functional preprocessing in the `LogFilter` module.

It is also recommended to expand the system's compatibility with more IDSs like Suricata and FortiGuard, including older versions of currently supported systems for backward compatibility, and other alert modes.

The virtual network translator requires generalization and optimization for diverse network settings. Moreover, in its present form, the translator is limited to supporting only iptables firewalls. This contrasts with the broader range of firewall types supported by VEREFOO and vlogi, which include Open vSwitch and EBPF among others. Future improvements should therefore focus on enhancing the translator to align with the capabilities of VEREFOO and vlogi.

Bibliography

[1] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Comput. Surv.*, vol. 56, no. 3, pp. 57:1–57:37, 2024. [Online]. Available: https://doi.org/10.1145/3616401

[2] 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

[3] ——, "Automated firewall configuration in virtual networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, 2023. [Online]. Available: https://doi.org/10.1109/TDSC.2022.3160293

[4] D. Bringhenti, R. Sisto, and F. Valenza, "A demonstration of VEREFOO: an automated framework for virtual firewall configuration," in *9th IEEE International Conference on Network Softwarization, NetSoft 2023, Madrid, Spain, June 19-23, 2023*, C. J. Bernardos, B. Martini, E. Rojas, F. L. Verdii, Z. Zhu, E. Oki, and H. Parzyjegla, Eds. IEEE, 2023, pp. 293–295. [Online]. Available: https://doi.org/10.1109/NetSoft57336.2023.10175442

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

[6] E. Karafili, F. Valenza, Y. Chen, and E. C. Lupu, "Towards a framework for automatic firewalls configuration via argumentation reasoning," in *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*. IEEE, 2020, pp. 1–4. [Online]. Available: https://doi.org/10.1109/NOMS47738.2020.9110399

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

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

[9] D. Bringhenti, R. Sisto, and F. Valenza, "A novel abstraction for security configuration in virtual networks," *Comput. Networks*, vol. 228, p. 109745, 2023. [Online]. Available: https://doi.org/10.1016/j.comnet.2023.109745

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

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

[12] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE J. Sel. Areas Commun.*, vol. 23, no. 10, pp. 2069–2084, Oct 2005.

[13] F. Valenza, C. Basile, D. Canavese, and A. Lioy, "Classification and analysis of communication protection policy anomalies," *IEEE/ACM Trans. Netw.*, vol. 25, no. 5, pp. 2601–2614, Oct 2017.

[14] C. Basile, F. Valenza, A. Lioy, D. R. Lopez, and A. P. Perales, "Adding support for automatic enforcement of security policies in nfv networks," *IEEE/ACM Trans. Netw.*, vol. 27, no. 2, pp. 707–720, Apr 2019.

[15] H. Yasser, "Verification and validation of automated policy enforcement in sdn and nfv in virtualized environment," Master's thesis, Politecnico di Torino, Turin, Italy, 2022. [Online]. Available: https://webthesis.biblio.polito.it/24644/

[16] F. Pizzato, "Optimized and automatic firewall reconfiguration," Master's thesis, Politecnico di Torino, Turin, Italy, 2023. [Online]. Available: https://webthesis.biblio.polito.it/26915/

[17] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Automatic and optimized firewall reconfiguration," 2024, accepted for publication in Proceedings of IEEE/IFIP Network Operations and Management Symposium 2024, NOMS 2024. [Online]. Available: https://hdl.handle.net/11583/2985072

[18] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," in *IEEE International Conference on Network Protocols (ICNP)*, 2013, pp. 1–11. [Online]. Available: https://doi.org/10.1109/ICNP.2013.6733614

[19] S. Bussa, R. Sisto, and F. Valenza, "Security automation using traffic flow modeling," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 486–491. [Online]. Available: https://doi.org/10.1109/NetSoft54395.2022.9844025

[20] "Docker Engine," https://docs.docker.com/engine/install/ubuntu/, Accessed: 2023-12-15.

[21] "Docker Compose standalone," https://docs.docker.com/compose/install/standalone/, Accessed: 2023-12-15.

[22] "Z3 Prover," https://github.com/Z3Prover/z3/releases/tag/z3-4.8.15, Accessed: 2023-12-15.

[23] "Cypher Shell," https://github.com/neo4j/cypher-shell/releases/download/1.1.15/cypher-shell_1.1.15_all.deb, Accessed: 2023-12-15.

[24] "Neo4j," https://go.neo4j.com/download-thanks.html?edition=community&release=3.5.25&flavour=deb, Accessed: 2023-12-15.

Appendices

Appendix A

Environment setup

This appendix provides guidance for future maintainers on setting up a development environment for this thesis work. It includes system requirements, mandatory dependencies, and optional dependencies that may be useful for future development of the VEREFOO framework.

A.1 System requirements

The development and testing of the process and related components were conducted on a VirtualBox VM configured as follows:

- Operating system: Ubuntu 20.04 LTS (64 bit)
- Processor: Intel Core i7-6700HQ CPU
- Storage: 50 GB HDD
- Memory: 4 GB RAM
- CPU cores: 4

A.2 Mandatory dependencies

The network created by the virtual network translator from a VEREFOO FAS is Docker-based. Both VEREFOO and VIP require Java 1.8, and VEREFOO additionally utilizes the *z3* theorem prover. Manual API requests to VEREFOO are made using `curl`, and vlogi requires Python's `watchdog` and `requests` packages. The following installations are necessary:

- **Docker Engine** - Follow the instructions provided at [20].
- **Docker Compose** - Install the standalone version as per instructions at [21].
- **OpenJDK 1.8** - Installation command is:

Listing A.1. Install OpenJDK 1.8

```
sudo apt install openjdk-8-jdk
```

- **curl** - Installation command is:

Listing A.2. Install curl

```
sudo apt install curl
```

- **Python's package manager** - Ubuntu 20.04 includes Python 3.8 by default. Install `pip` with:

Listing A.3. Install pip

```
sudo apt install python3-pip
```

- **Z3 Prover** - Download `z3-4.8.15-x64-glibc-2.31.zip` from [22]. Avoid newer versions as they may not be compatible with VEREFOO. Extract the archive, rename the folder to `z3`, and move it to `/home`. Then, add the following lines to `~/.bashrc`:

Listing A.4. Setting Z3's environment variables

```
LD_LIBRARY_PATH=/home/z3/bin/
Z3=/home/z3/bin/
```

A.3 Optional dependencies

These dependencies, while not required by the process developed in this thesis, could be valuable in the future. As some features of VEREFOO require Neo4j, and the included version with VEREFOO is Windows-specific, the steps below guide you in setting up Neo4j in the VM:

- Download and install `cypher-shell 1.1.15` from [23] and `neo4j 3.5.25` from [24]:

Listing A.5. Install Cypher Shell and Neo4j

```
sudo apt install ./cypher-shell_1.1.15_all.deb
sudo apt install ./neo4j_3.5.25_all.deb
```

  If links are broken, find the `.deb` files in the `neo4j-files` directory of the `ids-demo` branch of VEREFOO.

- Start Neo4j by running:

Listing A.6. Start Neo4j

```
sudo /usr/bin/neo4j console
```

- Access `http://localhost:7474/`, log in with username `neo4j` and password `neo4j`, then change the password to `costLess`.