Politecnico di Torino

Master's Degree in Cybersecurity

Master's Degree Thesis

An Automated Network Security Workflow based on OpenC2 and VEREFOO

Supervisors
prof. Daniele Bringhenti
prof. Fulvio Valenza
prof. Riccardo Sisto
dott. Gianmarco Bachiorrini

Candidate
Carlo Cimino

Academic Year 2024-2025

"Enjoy the butterflies, enjoy being naïve, enjoy the nerves, the pressure.
If you kind of want stand on the top from day one, then there's nothing else to look forward to.
Enjoy kind of the process of making a name for yourself and meeting some great people along the way. Theres a lot of worldly people that you can laugh with, learn from, enjoy some moments with.
So embrace the good ones, stay focused, don't be too far off your path, just keep trying to build and grow, and learn from yourself.
But don't forget what got you here.
Bring friends along, bring family along.
They might be something to take the weight off your shoulders, they're also people to enjoy the moments with, to celebrate with.
So don't be afraid to surround yourself with people that you care about and love, they're also so excited to be on this journey too.
Get after it!"

Summary

This thesis describes the design, implementation and validation of a connector that act as an automated bridge between the OpenC2 Context Discovery (CTXD) specification and the VEREFOO formal verification framework. The main objective is to create a closed-loop system for proactive security policy management in modern, dynamic network environments.
This thesis project addresses a fundamental challenge: bridging the structural gap between network context information, which is often non-standardized and the rigorous, structured input format required by formal verification tools. The connector is specifically built to manage real-time network topological data, provided in OpenC2 CTXD JSON format, and translate it into a comprehensive network graph model that is compliant with VEREFOO's NFV XML schema.
The implementation of the connector is detailed in a Python script that operate a multi-stage workflow. First, it performs data extraction and aggregation, parsing multiple JSON files to collect a holistic view of network services, links, and configurations. It then executes the data model mapping, which includes prioritizing IPv4 addresses for unique node identification, using heuristics to automatically deduce the functional role of each network element (e.g., FORWARDER, WEBSERVER), and ensuring graph coherence by creating nodes for implicit network elements and enforcing bidirectional link relationships.
Once the VEREFOO-compliant XML is generated, the connector automates its submission to the VEREFOO RESTful API. It is designed to handle the responses, employing a robust polling mechanism to retrieve the final verification results. The most critical function of the connector is the output analysis, where it parses the VEREFOO response, identifies policy violations flagged by an isSat="false" status, and translates them into precise, actionable OpenC2 commands. For example, a violated ReachabilityProperty triggers an allow command, while a violated IsolationProperty generates a deny command.
In summary, this thesis presents a solution that fills the critical gap between network discovery and security policy enforcement. It moves beyond traditional manual and reactive security measures by demonstrating an automated, continuous process where network context is dynamically discovered, the security provided by formally verification, and corrective actions are automatically generated. This approach significantly enhances network security by minimizing human error and providing a verifiable foundation for security policy compliance.

Acknowledgements

I would like to express my deepest gratitude to Professor Valenza, Professor Bringhenti, and Gianmarco Bachiorrini for their invaluable guidance and encouragement throughout the development of this master's thesis.
To my family, whose support, patience, and understanding have been the strength I relied on in every moment of difficulty. Their faith in me has been a constant source of courage.
A special thanks goes to Alessia, my girlfriend, whose love, support, and encouragement have accompanied me not only during this journey but in so many other moments of my life, making each step more meaningful.
My heartfelt thanks also go to my friends, who, despite the distance of more than a thousand kilometers, have always been close to me with their encouragement and affection.
Finally to my colleagues, whose collaboration and friendship created a stimulating and warm environment that made this experience not only more enriching but also more memorable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Keeping networks secure has become an increasingly complex and dynamic challenge in the modern cybersecurity landscape. The fast adoption of Software-Defined Networking (SDN) and Network Functions Virtualization (NFV), combined with the spreading of cloud services and IoT devices, has expanded the attack surface of cyber system [1]. In such environments, manually configuring security policies, particularly for critical functions like firewalls, is laborious, error-prone and often unfeasible tasks [2].
Traditional static security models and manual configuration processes already showed their increasingly inadequateness. Even a single misconfigured firewall rule can expose a critical vulnerability, potentially leading to data breaches, significant compliance penalties and lasting reputational damage to an organization. The volume of network traffic and the constant re-provisioning of virtual resources mean that a security analyst cannot keep up with the pace of change, creating a constant window of vulnerability.
This embedded vulnerability shows the need to shift from reactive and manual processes to a proactive and automated security approach. Attackers take advantage of the dynamic nature of virtualized networks to move in lateral ways, evading detection, targeting the supply chain, exploiting vulnerabilities in software components or hardware devices. These sophisticated, multi-stage attacks often exploit a series of small misconfigurations that would be impossible to identify and correct through manual inspection alone.
In addition to this complexity, there is the highly specialized nature of many security tools from different vendors, operating in isolation and being statically configured, creating a lack for common language to communicate and coordinate responses, disintegrating the cybersecurity operations [3]. Every individual siloed approach makes extremely difficult to achieve a holistic view of the network's security and to implement suited coordinated defenses, providing an asymmetric time advantage to the attackers. Without a standardized communication framework, the security ecosystem becomes a collection of fragmented and incompatible tools. An intrusion detection system might detect a threat, but it cannot automatically trigger a response from a firewall, if the components come from a different vendor without a custom integration.
The integration of different cybersecurity technologies requires coordinated actions, but this process is difficult due to the reliance on custom communication interfaces: these proprietary APIs make cross-platform integration costly and complex, also limiting the

ability to be dynamically integrated with other tools. While modern orchestration platforms try to solve this limitation with a plug-and-play architecture, the scalability issues still remain, invalidating the effectiveness that became based on the availability of the development of custom interfaces. Furthermore, while adopting technologies from multiple vendors can enhance security by preventing a single vulnerability from compromising an entire infrastructure (for example a zero-day vulnerability), this vendor diversity also complicates device management due to the fragmented nature of the defense.
Nowadays, the need for a proactive —rather than reactive— security stance is needed more than ever, requiring solutions that can adapt quickly to the changes and automatically ensuring their correctness. Automated responses must be not only fast, but also fundamentally correct. A fast, but flawed, automated action could potentially cause more harm than the original threat, such as by blocking legitimate business traffic or creating new vulnerabilities.
To overcome these challenges, automated and formally verifiable approaches to network configuration and security policy enforcement are game changer [4]. Standardizing information exchange, through mechanisms like OpenC2 Context Discovery, is needed for enabling this automation, allowing machines to communicate and share network context in a more efficient way.

1.1 The Role of OpenC2 in Cybersecurity Automation

To address these issues, the Open Command and Control (OpenC2) framework emerges as a key enabling technology. It provides a standardized language for setting up different cybersecurity tools, by defining a common syntax and structure for commands and responses, facilitating the automated exchange of information [1].
The OpenC2 language is built on the fundamental triad of "Action-Target-Arguments", which break down complex commands into three fundamental components, creating a common grammar for cybersecurity and allowing the understanding by any compliant system, regardless the vendor [3]. Actuator profiles extend the language to focus on specific cyber defense functions and provide conformance requirements for interoperability. This modular approach allows the language to be extended to new technologies without the need to change the standard, making it highly scalable and prone for future implementations. For instance, this capability is decisive for Context Discovery (CTXD), an OpenC2 Actuator Profile that allows a "Producer" to dynamically query "Consumers" (network devices or security tools) about their configuration details, like network's topology, active services, peers that are indispensable for any advanced automation and verification framework, as it provides the raw data needed to understand the current state of the network.

1.2 The Importance of Formal Verification in VEREFOO

The acquisition of dynamic network context, while central, is only one part of the solution. As anticipated before, the complexity of modern networks necessitates automation, but it should be a *verified* one. This is where formal verification tools, such as VEREFOO (VErified REFinement and Optimized Orchestrator), become indispensable.
Formal verification is a discipline eradicated in mathematics and computer science that is based on rigorous methods to prove the correctness of a system. The main difference with heuristic or simulation-based approaches is that they can only test a limited number of scenarios, which are based on a set of pre-defined rules that may not cover all possible cases; on the other hand, formal verification provides a mathematical guarantee that a system will behave as intended under all possible conditions.
In the context of network security, this means that a formally verified firewall configuration can be trusted to enforce policies without introducing unwanted vulnerabilities or unintentionally blocking legitimate traffic. This is particularly critical in dynamic environments where network state can change frequently, making it impractical to test every potential configuration manually.
VEREFOO leverages the rigorous formal methods by modeling the network and its security policies as a complex mathematical problem, a Maximum Satisfiability Modulo Theories (MaxSMT) problem, to mathematically guarantee the correctness and optimality of the generated firewall configurations. This provides a high level of assurance that the implemented security policies will work as intended. By translating the network's topology, services and security policies into a precise form, VEREFOO can exhaustively analyze all possible states and interactions, eliminating the risks associated with human error and incomplete testing.
Furthermore, the connector's ability to find the optimal configuration ensures that the security rules applied are not only correct but also efficient, avoiding the unnecessary ones that can reduce performance.

1.3 Covering the Operational Gap: the Thesis Objective

Despite the individual strengths of OpenC2 (dynamic context discovery and standardized command execution) and VEREFOO (formal verification and optimized policy generation), a significant operational gap exists between them [5]. OpenC2 provides structured information about the network's current state in a machine-readable JSON format. However, VEREFOO requires its input in a highly formalized, static XML schema, detailing network nodes, functional types and interconnections.
The main difficulties are translating the discovered context from OpenC2 into the precise semantically consistent and structured format required by VEREFOO.
A manual process is required to act as a bridge, where a security analyst must:

1. Manually parse the OpenC2 JSON response.

2. Identify the relevant network entities (nodes, links, services).
3. Map these entities to the specific fields and attributes required by VEREFOO's NFV (Network Functions Virtualization) XML schema.
4. Manually create the XML file, ensuring that all structural constraints (like node bidirectionality, functional types) are correctly specified.

This step is time consuming and could represent a bottleneck in a security workflow, that require an extremely fast response. As an example, it could be considered a scenario where a critical vulnerability is detected and requires a new firewall policy to be deployed immediately: if an analyst has to spend hours manually converting data, the organization remains exposed to the threat and the manual intervention introduces the risk of human error. OpenC2 and VEREFOO were built with the purpose of eliminating this issue.
This thesis addresses the critical integration gap, focusing on security policy management, and it aims to enable the full potential of complementary technologies by proposing a methodology and implementing a *connector*. This connector specifically converts network topological information and service capabilities, obtained via OpenC2 Context Discovery, into a specific XML format for formal verification using VEREFOO.

1.4 The *Connector* Implementation: Bridging the Gap

The connector's functionality goes beyond a simple data conversion. The script performs a crucial semantic analysis of the raw OpenC2 data to infer the "functional role" of each service, such as nat, webserver, webclient or forwarder This logic is essential because VEREFOO's formal model relies on these explicit functional types, which are not always present in the OpenC2 response. The connector acts as an intelligent interpreter, mapping the dynamic and often incomplete data into the precise format required for formal verification.
Another key aspect of the implementation is the handling of network addresses. Since VEREFOO requires a valid IP address for every node, the connector includes a dynamic IP assignment logic. For nodes that lack an explicit IPv4 address in the OpenC2 JSON, the script automatically assigns an arbitrary IP (starting from "20.0.0.1"), ensuring that every element in the network model can be correctly identified and handled within the formal model. Furthermore it automatically ensures the bidirectionality of links between nodes, a structural constraint necessary for VEREFOO's graph model, by adding a symmetric neighbour entry for each connection.
The connector also completes the automation loop by communicating to VEREFOO via REST API. After VEREFOO performs its verification and identifies any unsatisfied properties (REACHABILITY_PROPERTY or ISOLATION_PROPERTY), the script parses the output response provided by VEREFOO. It then interprets the verification results to generate executable OpenC2 commands (allow or deny rules) that correct the detected policy violations. This bidirectional process, from dynamic context discovery to policy enforcement commands, demonstrates the full potential of the integrated framework,

transforming a manual error-prone workflow into an automated formally verified security pipeline.
The central role of this connector in the overall security pipeline can be illustrated as a three-stage process:

1. Data Normalization: raw OpenC2 JSON data is cleaned and completed with semantic information.
2. Formal Modeling: the normalized data are converted into the precise XML format required by VEREFOO.
3. Communication with VEREFOO: the connector obtains the output from VEREFOO, if the verification result is 'SAT' (Satisfied).
4. Output Translation: the verified output is then translated back into actionable OpenC2 commands.

This holistic approach ensures that the entire security lifecycle —from network discovery to policy enforcement— is automated and verifiably correct.
The thesis is structured in different parts:

- Chapter 2 provides a comprehensive background on OpenC2 and VEREFOO, detailing their architectures and relevance to cybersecurity and for this thesis, establishing the foundations by explaining how these two frameworks enable automated security orchestration and formal verification.
- Chapter 3 defines the thesis objective, exposing the existing gap in the security management process, such as the lack of an automated and reliable bridge between OpenC2 Context Discovery data and VEREFOO's formal verification engine. Than, it outlines the objectives of this thesis, focusing on the design and implementation of a tool to fill this gap.
- Chapter 4 outlines a detailed overview of the system's architecture, describing the modular design of the connector. The chapter explains the data mapping strategy from OpenC2 JSON to VEREFOO's XML format and details the main logic that orchestrates API interactions, response parsing, and the generation of final OpenC2 Commands.
- Chapter 5 presents the validation methodology for the connector, demonstrating the correctness and effectiveness of the connector. It showcases four specific case study, illustrating every file obtained through the connector, which mean the XML file generated as output, the VEREFOO response file and the final OpenC2 Commands file. That provides concrete evidence of the tool's accuracy and effectiveness not only in preparing data for VEREFOO, but also in interpreting verification results and translating them into actionable security policies.
- Chapter 6 summarizes the conclusions drawn from this thesis project and discusses promising avenues for future work, considering the broader implications of the work.

Chapter 2

Background

This chapter provides a comprehensive overview of the two foundational concepts that are central to this thesis: OpenC2 and VEREFOO.
In today's dynamic threat landscape, cybersecurity tools often operate in isolated silos, hindering effective and timely responses to threats [2]. OpenC2, a standard for harmonizing Command-and-Control (C2) communication among cybersecurity tools, addresses this critical challenge by providing a common language that allows disparate systems to work together seamlessly. At the same time, network configurations are becoming increasingly complex, making them prone to misconfigurations and vulnerabilities that can be exploited by attackers. VEREFOO is a tool that leverages formal verification methods to rigorously analyze and validate network configurations, ensuring that they conform to a defined set of security policies.
By understanding the unique capabilities of both OpenC2 and VEREFOO, we can undersand the problem addressed in this thesis: bridging the gap between automated threat response and proactive security validation. These two technologies are the basis for understanding the problem domain and the solutions proposed in this thesis.

2.1 OpenC2

2.1.1 Cybersecurity Landscape Challenges

The current cybersecurity landscape is characterized by its increasing complexity and the continuous evolution of cyberthreats. Organizations face a growing volume and sophistication of attacks, while the proliferation of cloud adoption, virtualization and IoT devices continues to expand the attack surface. This dynamic environment places a lot of pressure on Security Operations Centers (SOCs) to respond rapidly and effectively to incidents. However, the different array of security products from various vendors leads to a fragmented ecosystem. Each tool typically exposes its own Application Programming Interface (API) and command-line interface (CLI), requiring costly, time-consuming and weak custom integrations [6]. This fragmentation makes coordinated responses hard and prone to human error, highlighting the urgent need for interoperable and automated

solutions.

2.1.2 Motivation and Principles of OpenC2

The escalating complexity and dynamic nature of modern cybersecurity landscapes require robust and interoperable solutions for threat response and system management. Standardized interfaces, protocols and data models are the key factors to enable continuous integration both within and across the various cyber defense systems. In this environment, the Open Command and Control (OpenC2) [7] [8] presents itself as a response, focused on the development of a language for inter operation between functional element of cyberdefence system; nowadays, it also became a standard that serves as a key enabler, providing a unified and vendor-agnostic language for managing different security tools, defining a set of abstract atomic cyberdefense actions in a platform [9].
OpenC2 is developed by an expert technical committee within the OASIS (Organization for the Advancement of Structured Information Standards) [3], to address the lack of standardized interfaces and command sets, providing a common ground for the smooth integration and automation of cyberdefense capabilities across the infrastructures of different organizations. By establishing common syntax and structure for commands and responses, OpenC2 facilitates the automated exchange of cyberdefense instructions, also enhancing reaction times, reducing manual intervention and improving overall security posture: all this is achieved through a modular architecture that allows the definition of "Actuator Profiles" —specifications tailored to specific classes of components— and the integration of different protocols. Semantic clarity guarantees that commands are unambiguous and can be interpreted consistently across different implementations, minimizing the risk of misconfiguration or unintended actions. By standardizing the "what" (actions and targets) and decoupling it from the underlying "how", OpenC2 allow a smooth machine-to-machine communication for cyberdefense, providing also a greater interoperability and automation within heterogeneous security ecosystems.
However, it is important to recognize that while a language like OpenC2 is necessary for coordinated cyber responses, it is not enough on itself. Critical aspects of the selection of appropriate courses of action fall outside the scope of OpenC2 and must be addressed through complementary technologies and strategies within a comprehensive cybersecurity framework. This separation of concerns is a foundational principle of OpenC2: it defines the "language" for communication but leaves the "decision-making logic" to an external orchestrator or a human analyst. This makes the standard very flexible, allowing it to be integrated with a wide range of analysis and verification tools. For example, a threat intelligence platform can decide to block a malicious IP address and an OpenC2 command can then be used to communicate this decision to a firewall, regardless of the firewall's vendor.
The design of OpenC2 is based on four key principles [3]:

- Technology agnostic: it defines abstract and atomic actions to ensure interoperability regardless of underlying implementations.

- Concise: OpenC2 language is pretty minimal, with a deep focus on essential information required to minimize communication overhead in network-constrained environments.
- Abstract: Commands and Responses are abstracts, allowing the encoding and the transferring via different schemes.
- Extensible: created in a modular design to evolve beside cyberdefense technologies, it can be extended for introducing future cyberdefense functionalities.

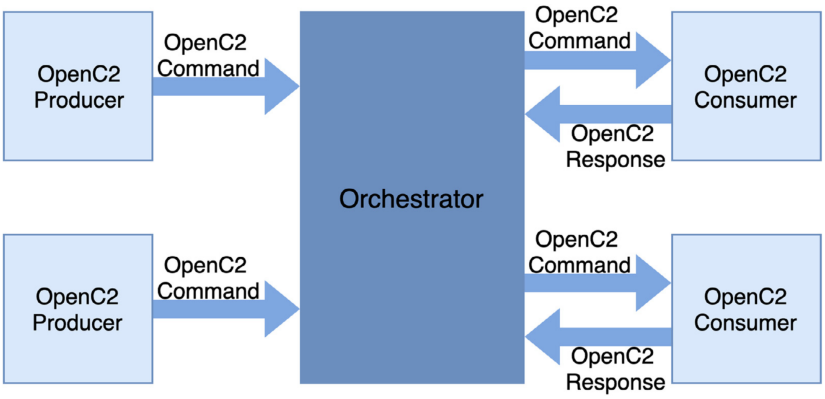

Figure 2.1. Architecture with OpenC2 interfaces

2.1.3 OpenC2 Language Structure and Components

As widely explained in [7], an OpenC2 message typically comprises a Command and a Response. It uses a request-response paradigm, where a Command is encoded by a Producer (who manage the application), and the Consumer send a Response with status and the requested information.
The two participants, as just mentioned, are:

1. **Producer**: it creates and sends Commands instructing one or more systems to execute specified Actions; it may be a SOAR (Security Orchestration, Automation and Response) platform, or a threat intelligence feed.
2. **Consumers**: it receives and acts upon a Command, may create Responses to provide the information captured or necessary to response to the Producer; it may be a firewall, a intrusion detection system (IDS) or a network sensor.

OpenC2 **Commands** are instructions defined by an action-target pair and are used by the Producer to perform specific Actions, received by the Consumer and executed by the Actuator, to carry out the Target and may optionally include details as identifier, parameters or specified Profile [7]. A brief definition of the four fields is reported below:

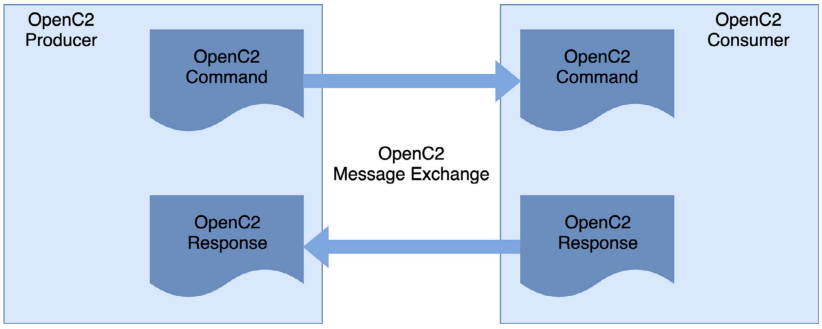

Figure 2.2. OpenC2 Message Exchange

- **Action** (mandatory): defines the operation that has to be executed by an actuator.
  - *scan, locate, query, allow, deny, contain, start, stop, restart, set, update, ...*
- **Target** (mandatory): identifies what is acted upon.
  - *artifact, command, device, domain_name, email_addr, ipv4_connect, ipv4_net, mac_addr, uri, ...*
- **Arguments** (optional): provide execution details and granularity, such as timing, duration, periodicity, need of acknowledgment, additional status information.
  - *start_time, stop_time, duration, response_requested, comment*
- **Profile** (optional): indicates the function to be performed and ensures compatibility for Consumers, else the Command will be ignored.
  - *slpf, sfpf, pf, ids, av, hp, er, log*

OpenC2 **Responses** are used as information return by a Consumer, to the Producer, as a result of the execution of a Command, and serve as acknowledgment receipt, return execution status and other relevant data. They are composed by:

- **Status** (mandatory): indicates the outcome of the Command.
  - *102 - Processing, 200 - OK, 400 - Bad Request, 401 - Unauthorized, 403 - Forbidden, 404 - Not Found, 500 - Internal Error, 501 - Not Implemented, 503 - Service Unavailable*
- **Status text** (optional): describes the status code.
- **Response** (optional): additional information.
  - *versions, profiles, pairs, rate_limit*

The example shows an `OpenC2 Command` for a StateLess Packet Filter (SLPF), allowing an IPv6 connection, using the common JSON (JavaScript Object Notation) serialization:

```
{
  "action": "allow",
  "target": {
    "ipv6_connection": {
      "protocol": "tcp",
      "dst_addr": "3ffe:1900:4545:3::f8ff:fe21:67cf",
      "src_port": 21
    }
  },
  "actuator": {
    "slpf": {}
  }
}
```

This example shows the `OpenC2 Response`, in which the Actuator returns a rule number associated with the allowed interaction:

```
{
  "status": 200,
  "results": {
    "slpf": {
      "rule_number": 1234
    }
  }
}
```

2.1.4 Implementation

OpenC2 implementations incorporate the OpenC2 core specifications with relevant industry standards, protocols and other specifications [7]. The Fig. 2.3 outlines the links within the OpenC2 specification family, with the external environment-specific deployments. It is important to highlight that the layering implementation is notional and is not a restrict specific approach (such as utilizing application-layer mechanisms for message authentication and integrity).
OpenC2 can be conceptually partitioned in four layers:

- Function-Specific Content: defines the language elements deployed to provide a cyberdefense function.
- Common Content: defines the Commands and Responses structure and a set of common language used to create them.
- Message: defines a transfer and a content mechanisms for the transmission of Messages.

- Secure Transport: provides the path of communication between the Producer and the Consumer.

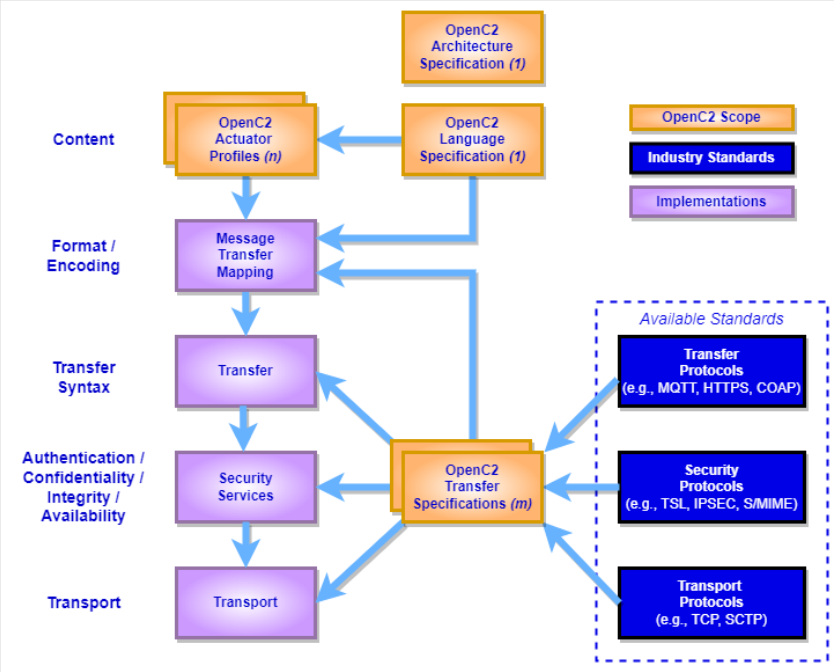

Figure 2.3. OpenC2 Model

As shown in [8], OpenC2 data types are defined using the types and options available in JADN (JSON Abstract Data Notation) [10]. It is a UML-based information modeling language which defines data structure independently of data format, providing a tool used to define and generate physical data models, validate information instances and enable lossless translation across data formats. By using JADN, it is possible to define types that are decoupled from their internal representation within applications (known as "API" values) and the format used for transmitting data between systems ("serialized" values):

- API values: must adhere the characteristics of the defined type, while their representation does not impact interoperability.
- Serialized values: ensure interoperability, have specified unambiguous serialization rules for all the possible types, defined in JSON (JavaScript Object Notation) format.

The standard `query features` command in OpenC2 allows a Producer to understand the Actuator's capabilities, but it does not provide the important contextual information about the nodes and the interconnections within the network.
To solve this lack, as described in [11], a new Actuator Profile for the OpenC2 language has been developed: the Context Discovery (CTXD). This profile is specifically designed

to enable an OpenC2 Producer to discover and understand the environment, including running services, their interactions and all the integrated security features. By implementing CTXD Actuator Profile, the Consumers can provide specific and detailed contextual information, allowing the Producer to build a precise network's topology. The main goal of CTXD is to verify a service's type, characteristics, important connectivity details like hostname, encoding format and transfer protocol. This creates an operational context for each service within the network.
The steps of the pipeline are outlined as follow:

- a Producer dispatches an OpenC2 `query` command specifically targeting CTXD Actuator.
- If, and only if, the Consumer supports this profile, it processes the command by employing a sophisticated recursive function.
- This function triggers a set of iterative queries trying to discover new digital resources and to obtain their features, helping to construct a detailed map of the network's services and their interdependencies.

Context Discovery (CTXD) Actuator Profile

As accurately explained by [11], while OpenC2 provides a unified language for cyber defense, a big challenge in modern cybersecurity remains the fragmented ecosystem of security tools and the inherent lack of real-time knowledge about the dynamic network environment. The standard `query features` command in OpenC2 allows a Producer to understand an Actuator's capabilities, but it does not provide the crucial contextual information about the services, devices, and their interconnections within the network. This gap in visibility hinders the ability of Security Operations Centers (SOCs) to respond rapidly and effectively to complex incidents with fully informed, automated actions. To address this critical need for greater situational awareness, a new Actuator Profile for the OpenC2 language has been developed: the Context Discovery (CTXD). This profile is specifically designed to enable an OpenC2 Producer to discover, abstract and understand the dynamic state of the operational environment, including running services, their interactions, and integrated security features. By implementing CTXD Actuator Profile, Consumers can provide detailed contextual information, allowing the Producer to build a comprehensive map of the network's topology and its security landscape. The core functionality of CTXD revolves around identifying digital resources and their relationships. It systematically ascertains a service's type and characteristics, along with essential connectivity details such as hostname, encoding format and transfer protocol. This establishes a holistic operational context for each service within the network. From an operational standpoint, a Producer dispatches an OpenC2 `query` command specifically targeting CTXD Actuator. If the Consumer supports this profile, it processes the command by employing a sophisticated recursive function. This function iteratively queries newly discovered digital resources to extract their features, thereby constructing a detailed map of the network's services and their interdependencies.

OpenC2 Command Extensions

The Context Discovery profile expands the basic OpenC2 language elements to build the needed requirements of context discovery, sticking to the principles of OpenC2's architecture. This ensures that CTXD remains a passive, information-gathering mechanism, clearly distinct from active command and control actions like deny or allow.

- **Action**: CTXD profile uses only the `query` action, as the Actuator Profiles extend the OpenC2's functionality and does not have to introduce new ones.
- **Target**: there are main modifications used to request information on active services and their links, expanding the optional fields:
  - `services`: a list of service names for which detailed information are requested. If empty, it is a request for all discovered services.
  - `links`: a list of link names for which details are searched. If empty, it is a request for all discovered links.
  - `response_requested`: not new, indicates the desired level of response *(none, ack, status, complete)*.
  - `name_only`: boolean flag that, if *true*, imposes if the Consumer returns only the names of services and links; else, it provides the full detailed objects.
- **Arguments**: expand the standard arguments *(start_time, duration)* with the `name_only` argument is a command argument.
- **Profile**: the header must explicitly specify `ctxd`, assuring that the Consumer interprets correctly the command in the Context Discovery framework.

OpenC2 Response Extensions

The Context Discovery profile expands the basic OpenC2 language elements, while maintaining the standard OpenC2 response structure (including *status* and *status_text*), CTXD introduces a dedicated field within the results section to deliver the discovered information.

- **Results**: includes an `x-ctxd` field (custom extension, denoted by `x-`) which contains the specific output of the Context Discovery operation. It is structured to return the requested services and links, can be as full objects or only their names, based on the flag defined in `name_only` that originate the command:
  - `services`: list of Service objects, provide comprehensive details about each discovered service *(populated if name_only is false)*.
  - `links`: list of `Link` objects, defining the connections between services *(populated if name_only is false)*.
  - `services_names`: list of `Name` objects, containing only the names of the discovered services *(populated if name_only is true)*.

  - `link_names`: list of `Name` objects, containing only the names of the discovered links *(populated if name_only is true)*.

The `services` and `links` fields are mutually exclusive with `services_names` and `link_names`.

CTXD Data Types

The Context Discovery profile defines a wide set of data types, based on JADN, to precisely represent various aspects of the network context:

- Name: a `Choice` type used to identify any object in the context: `uri`, `reverse_dns`, `uuid`, `local`.
- Operating System (OS): a `Record` detailing an operating system with attributes: `name`, `version`, `family`, `type`.
- Service: a `Record` type that provide the services' information requested by the Producer: `name`, `type`, `links`, `subservices`, `owner`, `release`, `security_functions`, `actuator`.
- Service-Type: a `Choice` type that specify the general category: `application`, `vm`, `container`, `web_service`, `cloud`, `network`, `iot`.
- Application: a `Record` for generic software applications: `description`, `name`, `version`, `owner`, `type`.
- VM (Virtual Machine): a `Record` that details a Virtual Machine: `description`, `id`, `hostname`, `os`.
- Container: a `Record` for generic software containers: `description`, `id`, `hostname`, `runtime`, `os`.
- Web Service: a `Record` describing a generic web service: `description`, `server`, `port`, `endpoint`, `owner`.
- Cloud: a `Record` for generic Cloud services: `description`, `id`, `name`, `type`.
- Network: a `Record` describing a generic network service: `description`, `name`, `type`.
- Network-Type: a `Choice` specifying various network service types: `ethernet`, `802.11` (Wi-Fi), `802.15` (WPAN), `zigbee`, `vlan` (Virtual Local Area Network), `vpn` (Virtual Private Network), `lorawan`, `wan` (Wide Area Network).
- IOT (Internet of Things): a `Record` for IoT devices: `description`, `name`, `type`.
- Link: a `Record` that describes the type of connection between `Service` entities and any security features applied directly to the link: `name`, `description`, `versions`, `link_type`, `peers`, `security_functions`.

- Link-Type: an `Enumerated` type that describes the type of the link between a peer and the service under analysis: `api`, `hosting`, `packet_flow`, `control`.
- Peers: a `Record` type (object) used for the iterative discovery process: `service_name`, `role`, `consumer`.
- Peer-Role: an `Enumerated` type that defines the role of the `Peer` connected with the service under analysis: `client`, `server`, `guest`, `host`, `ingress`, `egress`, `bidirectional`, `control`, `controlled` .
- Consumer: a `Record` that provides the needed networking parameters for the connection to an OpenC2 Consumer: `server`, `port`, `protocol`, `endpoint`, `transfer`, `encoding`.
- Server: a `Choice` that specifies information of a server: `hostname`, `ipv4-addr`.
- Transfer: an `Enumerated` type defining the transfer protocol, can be extended with other transfer protocol: `http`, `https`, `mqtt`.
- Encoding: an `Enumerated` type that defines the encoding format for messages, can be extended to include other encoders (like XML): `json`.
- OpenC2-Endpoint: a `Record` type corresponding directly to an OpenC2 Actuator Profile and the endpoint that implements it: `actuator`, `consumer`.

OpenC2 Transfer Specification HTTPS

The OpenC2 Transfer Specification for HTTPS is a dedicated standard used to formalize the secure transport layer for every OpenC2 messages, establishing a robust client-server communication model where the OpenC2 Producer acts as an HTTP client and the OpenC2 Consumer functions as an HTTP server. The main reason why HTTPS is used is to ensure that all the message exchanges are secured via a TLS tunnel (using TLS 1.2 or a higher version) and secure cipher suites. Also, to provide cryptographic authentication of both the Producer and Consumer, the specification requires mutual TLS (mTLS, Transport Layer Security) to provide cryptographic authentication of both the side, which is the basis for trusted Command and Control. The OpenC2 Commands are transmitted using the HTTP POST method, with the command payload, serialized in JSON containing the body of the HTTP Request; the same occurs for the Responses, that are returned within the HTTP response body. A key feature of this layered approach is the clear separation of concerns: standard HTTP headers convey transport-level metadata and status, while the OpenC2 message itself, identified by the application/openc2+json content-type, handles the command and response logic. This also includes using HTTP status codes (for example 200 OK, 400 Bad Request), which indicate the success or failure of the message delivery, that is different from the OpenC2 status field which reports on the success of the Command execution. Considering these precise mechanisms, the specification guarantees reliable, secure and interoperable communication for security automation [12].

The following Fig. 2.4 illustrates the Producer/Consumer interactions. The process begins when the Producer, that need to send OpenC2 Commands, initiates a TCP (Transmission Control Protocol) connection to the Consumer: after the TCP handshake, a TLS session is established to mutually authenticate both endpoints and ensure the confidentiality of the connection. Within this secure tunnel, the Producer transmits OpenC2 Commands as HTTP requests using the `POST` method, with the Consumer's OpenC2 Responses returned in the corresponding HTTP response.

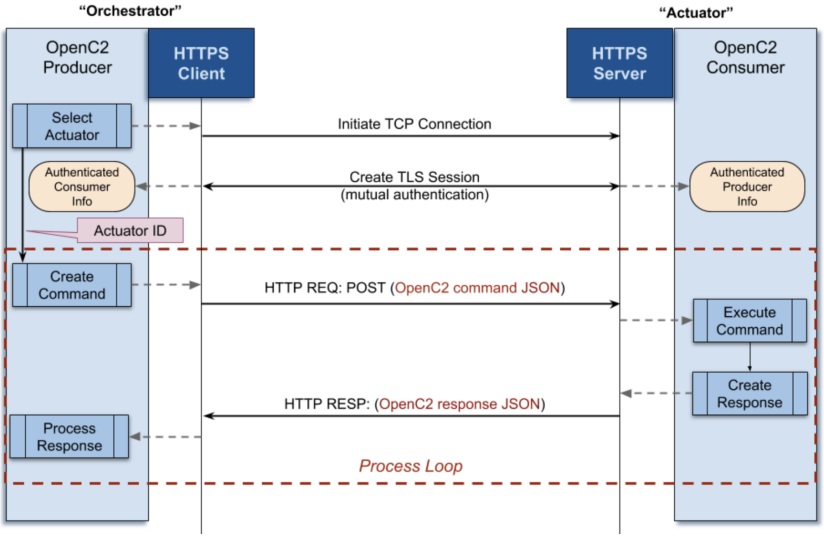

Figure 2.4. OpenC2 Interactions

Example of OpenC2 JSON Response

One of the most complete example to illustrate the structure of a Response received using the Context Discovery profile to OpenC2 is the *kube0.txt* file. This file shows all the field and information about the services and links discovered in the network:

```
{
   "headers": {
      "request_id": "d2f71ea6-a09a-4d41-a62f-0f8bbcf15c08",
      "created": 1743783591683,
      "from": "testconsumer",
      "to": [
         "producer.example.net"
      ]
   },
   "body": {
      "openc2": {
         "response": {
```

```
"status": 200,
"status_text": "OK",
"results": {
   "x-ctxd": {
      "services": [
         {
            "name": {
               "local": "vm"
            },
            "type": {
               "vm": {
                  "description": "vm",
                  "id": "b1820977-693d-4eef-9f0a-cd707343478b",
                  "hostname": "kube0",
                  "os": {
                     "name": "Debian GNU/Linux 12 (bookworm)",
                     "family": "linux"
                  }
               }
            },
            "links": [
               {
                  "local": "1821c3e4-584c-4e9a-b074-c35e9084ee6a"
               },
               {
                  "local": "b1374852-ba74-46be-901f-d08b19190892"
               },
               {
                  "local": "kubernetes"
               },
               {
                  "local": "os-fw"
               }
            ],
            "owner": "openstack",
            "actuator": {
               "server": {
                  "hostname": "kube0"
               },
               "port": 8080,
               "protocol": "tcp",
               "endpoint": "/.well-known/openc2",
               "transfer": "http",
               "encoding": "json"
            }
```

```
      }
   ],
   "links": [
      {
         "name": {
            "local": "1821c3e4-584c-4e9a-b074-c35e9084ee6a"
         },
         "link_type": "hosting",
         "peers": [
            {
               "service_name": {
                  "local": "container\n10.17.0.233"
               },
               "role": "guest",
               "consumer": {
                  "server": {
                     "hostname": "udr-0"
                  },
                  "port": 8080,
                  "protocol": "tcp",
                  "endpoint": "/.well-known/openc2",
                  "transfer": "http",
                  "encoding": "json"
               }
            }
         ]
      },
      {
         "name": {
            "local": "b1374852-ba74-46be-901f-d08b19190892"
         },
         "link_type": "hosting",
         "peers": [
            {
               "service_name": {
                  "local": "container\n10.17.0.234"
               },
               "role": "guest",
               "consumer": {
                  "server": {
                     "hostname": "upf-0"
                  },
                  "port": 8080,
                  "protocol": "tcp",
                  "endpoint": "/.well-known/openc2",
```

```
                "transfer": "http",
                "encoding": "json"
            }
        }
    ]
},
{
    "name": {
        "local": "kubernetes"
    },
    "description": "kubernetes",
    "link_type": "hosting",
    "peers": [
        {
            "service_name": {
                "local": "kubernetes"
            },
            "role": "guest",
            "consumer": {
                "server": {
                    "hostname": "kubernetes"
                },
                "port": 8080,
                "protocol": "tcp",
                "endpoint": "/.well-known/openc2",
                "transfer": "http",
                "encoding": "json"
            }
        }
    ]
},
{
    "name": {
        "local": "os-fw"
    },
    "description": "slpf",
    "link_type": "protect",
    "peers": [
        {
            "service_name": {
                "local": "slpf"
            },
            "role": "control",
            "consumer": {
                "server": {
```

```
                                "hostname": "os-fw"
                            },
                            "port": 8080,
                            "protocol": "tcp",
                            "endpoint": "/.well-known/openc2",
                            "transfer": "http",
                            "encoding": "json"
                        }
                    }
                ]
            }
        ]
    }
  }
}
```

2.2 VEREFOO

2.2.1 Evolution of Network Management

Network architecture has evolved from static networks to highly dynamic Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) environments, introducing flexibility in the network.
In the past, firewalls were the single points of control between local and external networks, but nowadays SDN and NFV enable a wider distribution across various service functions, demanding more sophisticated and automated deployment strategies to maintain efficiency and scalability. The combinations of these technologies enable the design of Service Graphs (SGs) that define the functions and their connections in a network service [13].
As widely explained before, an automated approach would facilitate the deployment and setup of Network Security Functions (NSFs), where actually are slowed down by manual tasks and tendency to errors. These NSFs are essential in enforcing Network Security Requirements (NSRs), which define the necessary security behaviors of the network.
An easy example is the necessity to isolate a compromised node after an attack: on one hand, if the task is performed manually, can be time-consuming, error-prone and maybe ineffective [14]; on the other hand, automation allows a precise and efficient placement of the NSFs required, ensuring correctness and optimality. Formal correctness is an additional advantage as it removes the necessity of manual verification, basing the effectiveness on a mathematical model [15]

2.2.2 VEREFOO Overview

To overcome these challenges, VEREFOO, which stands for VErified REFinement and Optimized Orchestrator, has been designed to provide an automated and formally verified method for allocating packet filters in a Service Graph and automatically configuring their rules, based on input specified security requirements. This framework solves the problem of manual configuration, guaranteeing the correctness of the generated firewall policies and optimizing the allocation scheme and the number of rules.
The framework takes as input the Service Graph describing the virtual network topology and a set of Network Security Requirements (NSRs), which it then uses to formulate a partial weighted *Maximum Satisfiability Modulo Theories* (MaxSMT) problem. This problem is solved to find an *optimal* allocation scheme and configurations of packet filtering firewall [16]. This ensures formal correctness, meaning that all security requirements are provably satisfied, and optimized deployment, minimizing the number of required firewall number and rules [1]: by relying on a rigorous formal model, VEREFOO inherently ensures the 'correctness-by-construction' of the proposed solutions, eliminating the necessity of post-deployment formal verification [17]. The real innovation lies in a new network modeling approach: instead of working on individual packets, VEREFOO models "packet classes", significantly improving the performance and scalability of the verification process, especially in complex virtualized networks [18]. This approach guarantees mathematical rigor without compromising efficiency [19]. The term optimal refers to a combination of:

- minimizing the number of allocated firewall in the SG;
- minimizing the number of rules inside each firewall, storing in a smaller portion possible of memory and, at the same time, improving the filtering operations' performances.

The MaxSMT solver works to find a solution that satisfies all the mandatory "hard" clauses while maximizing the satisfaction of "soft" clauses, which represent the optimization goals. This approach effectively transforms a complex network design problem into a solvable mathematical one, guaranteeing that if a solution exists, it is both correct and optimized according to the defined criteria. This is in line with [19], who propose an optimized approach for network security policy enforcement in complex SDN/NFV environments, highlighting the necessity of automated methods to manage evolving cyber threats which is in clear contrast to heuristic-based approaches which might find a good solution but offer no guarantees on its correctness or optimality.

2.2.3 VEREFOO Architecture

As described in [20], the VEREFOO framework is organized into three main phases, that automate the process from the security specification to the policy deployment:

- First phase, Network Description and Intent Specification: this is the only phase that requires active human interaction, in fact the user provides the two inputs required from the framework, namely the Service Graph describing the network topology and the Network Security Requirements.

- Second phase, Firewall Allocation and Configuration: this phase begins when VEREFOO receives the inputs in an XML file. The framework internally validates the inputs against the NFV XML schema to ensure correctness before proceeding. Then it formulates the MaxSMT problem, which is the core functionality of VEREFOO. The problem is solved using a state-of-art MaxSMT solver, specifically the Z3. The output in this phase is an XML file describing the optimized firewall allocation and configuration in a vendor-agnostic syntax.

- Third phase, Low-Level Rules Generation: the generic output rules are converted into the specific syntax of the actual firewall deployed in the network (like iptables), translating the generic rules into concrete configurations that can be passed to an orchestrator and than being deployed on actual firewall systems.

The actual work tested in this thesis is focused on the second phase, while the first and the third will be implemented as part of this thesis project.

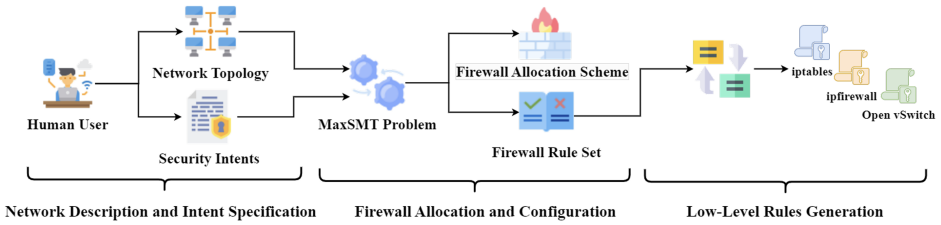

Figure 2.5. VEREFOO Architecture

2.2.4 Problem Formulation as MaxSMT

To formulate the problem as a partial weighted Maximum Satisfiability Modulo Theories (MaxSMT) problem, the two required inputs are a graph, that describes the network functions and the interconnection, and a set of NSR. The problem receives two sets of clauses:

- Hard clauses, which must be necessarily satisfied, represent the fundamental input of the problem, to reduce the space of possible solutions;

- Soft clauses are relaxable constrains, where each one has an assigned weight, representing the firewall positioning in the graph and the configurations, and this maximization contributes to the optimal solution.

If at least one hard clause can not be satisfied, the MaxSMT problem outcome is "UNSAT". Conversely, if all hard constraints can be satisfied, the result is the optimal firewall allocation scheme and the automatically computed Filtering Policy (FP) configuration, achieving the stated optimization requirements.
The already cited Service Graph is a directed graph where nodes represent basic network elements or endpoints, like terminals or physical servers, and edges define the network

links between them. SG acts as an abstraction, separating logical network from physical structure. From the SG, an internal structure called the *Allocation Graph* (AG) is automatically derived by adding *Allocation Places* (APs) to every edge, which are basically some potential positions for placing firewalls, to explore all possible placements and ensure the desired optimality [1]: the service designer may restrict or enforce placements through constraints, for example either forbidding APs in specific positions or requiring them where certain VNFs are already deployed, allowing fine-grained control over firewall placement [17]. It is important to underline that AG is also a directed graph, but its node set includes the original endpoints and service functions plus the APs.
To each node and link in the graph, a unique identifier, which is crucial for the MaxSMT solver to precisely track network elements and enforce policies at specific locations. For example, a `functional_type` attribute in the input XML file helps the system understand the role of each node (WEBSERVER, WEBCLIENT, NAT), which informs the logical model and the potential for applying security policies.

2.2.5 Security Requirements Specification

With the Allocation Graph (AG) formally defined, the next crucial input expected to the MaxSMT is the set of Network Security Requirements (NSRs), which defines the specific connectivity constraints that have to be enforced in the network. Each NSR reflects a precise security policy: either a *reachability* requirement, where communication must be allowed, or an *isolation* one, where communication must be blocked by the firewall.
The methodology includes a *general behavior* that is applied to all unspecified flows which do not match the specific requirements, creating a minimum of security in the system[1].
To define the set of security constraints, four approaches are supported for specifying the required security constraints:

- Whitelisting: the default behavior is to block all traffic flows, except for the explicitly allowed ones.

- Blacklisting: the default behavior is to allow all traffic flows, except for the explicitly denied ones.

- Rule-oriented Specific: allows to explicitly formulate both isolation and reachability properties, without defining a default behavior, hence the system automatically chooses how to manage the unspecified traffic depending on its specific goals. The main goal is to minimize the number of rules.

- Security-oriented Specific: as the rule-oriented specific approach, it allows both isolation and reachability properties. Its objective is to permit only the strictly necessary communications required to satisfy all user requirements.

To avoid ambiguity and ensure deterministic behavior, this set of NSRs is assumed to be conflict-free, since it is obtained using well-established conflict detection techniques, eliminating the need for rule prioritization[17].

2.2.6 Firewall Allocation and Policy Generation

Formally, each NSR is modeled as a 6-tuple (*type, IPsrc, IPdst, psrc, pdst, proto*) where the *type* identifies whether the rule enforces reachability or isolation, while the remaining fields specify the relevant packet header values (IP source, IP destination, port source, port destination, protocol): these rules form the hard clauses in the MaxSMT problem and strictly limit the solution space.
If the MaxSMT problem is satisfiable:

- The first output generated is the *optimal* Allocation Scheme of the firewall instances in the SG. This outcome is derived by evaluating every available Allocation Place (AP) as a candidate for firewall deployment and, to minimize resource usage, the system adopts solutions that allocate the fewest possible firewalls. As already explained, to achieve this, the placement of each firewall is modeled as a soft constraint whose satisfaction is weighted and evaluated.

- The second output is the configuration of the Filtering Policies (FP) for the deployed firewalls: each firewall policy includes a default action and a set of specific rules based on the 5-tuple structure of IP-based traffic. The *default* action is strategically chosen to reduce the number of required filtering rules, depending on the approach adopted for defining the NSRs.

For each firewall instance deployed at a given AP, the system identifies a set of *placeholder rules* that represents the maximum number of entries that may be needed in the policy. To avoid generating unnecessary rules, two main strategies are applied: if a specific NSR relates to a traffic flow that can not pass through the AP under consideration, no rule is created for it at that location; if the default behavior of the firewall already satisfies the requirement (like isolation requirement in a whitelisting-based configuration), there is no need to add a specific rule for that traffic flow.
If the MaxSMT problem is unsatisfiable, as cited before, the system generates a non-enforceability report, allowing the analysis on the reason why the given NSRs could not be satisfied. One common reason is that the Service Graph (SG) does not include sufficient Allocation Places (APs) for firewall placement, usually because the overly restrictive constraints.

2.2.7 Rule Minimization and Wildcard Usage

To further reduce the number of placeholder rules, the *wildcards* feature are used, which allow the aggregation of multiple NSRs into a single rule by generalizing IP addresses, ports or protocol fields: for example, instead of writing distinct rules for each individual IP in a subnet, the use of a wildcard rule can cover the entire range, as long as it does not compromise the satisfiability of other constraints.
After the definition of this rules, the additional soft constraints are defined to guide the configuration of each firewall:

- The first set focuses on minimizing the number of rules that are actually implemented, prioritizing configurations where placeholder rules are left unused: for doing

that, the associated weights are assigned in a way that reflects a higher penalty for adding rules than for using wildcards.

- The second set of soft constraints encourages the use of wildcards within the individual fields of each rule (*IPsrc, IPdst, psrc, pdst, proto*), keeping a more compact filtering policy and also improving processing efficiency. Nevertheless, the system ensures that using a wildcard remains a lower-priority objective compared to removing rules.

Finally, the entire set of constraints (hard and soft) is passed to the MaxSMT solver. If a solution exists (SAT), it provides the optimal firewall deployment and the corresponding configuration policies that respect all the defined security requirements while achieving minimal resource usage. If instead the solver finds the problem unsatisfiable (UNSAT), it returns a non-enforceability report, which indicates that the current set of constraints, and any limitations placed on AP usage, prevents a valid solution from being found. The output can then be used to adjust the input parameters and rerun the optimization process under revised conditions.
The core of VEREFOO's output is an XML file that includes the final configuration policies for each deployed firewall. The connector parses this XML to translate the abstract IsolationProperty and ReachabilityProperty into concrete network commands. For instance, a property with name="IsolationProperty" src="10.0.1.1" dst="10.0.2.1" would be translated into a deny action targeting the specified IP addresses. This translation is essential for enabling true automation and bridging the gap between the formal model and the real-world network deployment.

2.2.8 Example Scenario

To illustrate the practical application and efficacy of VEREFOO [1], let's consider Fig. 2.6, which shows the Allocation Graph (AG) derived from a Service Graph (SG) where endpoints such as e3 and e4 represent subnetworks rather than individual hosts.

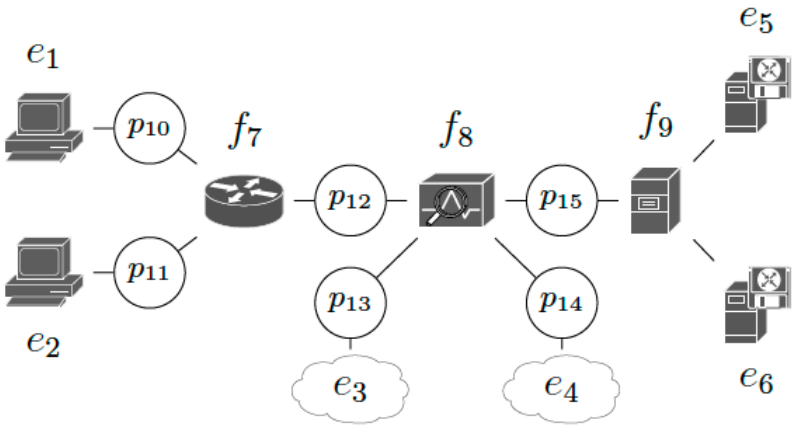

Figure 2.6. Before: Allocation graph

The Table 2.1 provides the mapping between SG nodes and their corresponding IP addresses or subnets along with their function types, while Table 2.2 provides the set of Network Security Requirements (NSRs) defined using the specific approach.

Identifier	IP address	Function type
e1	192.168.0.2	Web client
e2	192.168.0.3	Web client
e3	130.192.225.*	Network of end points
e4	130.192.120.*	Network of end points
e5	220.226.50.2	HTTP web server
e6	220.226.50.3	POP3 mail server
f7	120.0.2.2	NAT
f8	120.0.2.3	Traffic monitor
f9	120.0.2.4	Web cache

Table 2.1. IP Addresses and Function Type

Type	IPSrc	IPDst	pSrc	pDst	tProto
Isol	192.168.0.2	220.226.50.*	*	*	*
Isol	192.168.0.3	220.226.50.*	*	*	*
Isol	130.192.225.*	220.226.50.3	*	*	*
Reach	130.192.225.*	220.226.50.2	*	80	TCP
Isol	130.192.225.*	220.226.50.2	*	!=80	TCP
Isol	130.192.225.*	220.226.50.2	*	*	UDP
Reach	220.226.50.2	130.192.225.*	*	*	*
Isol	130.192.120.*	220.226.50.2	*	*	*
Reach	130.192.120.*	220.226.50.3	*	110	TCP
Isol	130.192.120.*	220.226.50.3	*	!=110	TCP
Isol	130.192.120.*	220.226.50.3	*	*	UDP
Reach	220.226.50.3	130.192.120.*	*	*	*
Isol	130.192.120.*	130.192.225.*	*	*	*

Table 2.2. Network Security Requirements

Initially, the focus is on the first two NSRs, which require isolating endpoints e1 and e2, that are hidden behind NAT f7, from services e5 and e6. To satisfy these, the system needs at least one firewall: given APs p10, p11, and p12, the optimal solution is to deploy a whitelisting firewall at p12, that intercepts all outbound traffic from the NAT. This strategy minimizes resource usage, in contrast to a manual approach that might incorrectly place two separate firewalls at p10 and p11.
Continuing to other NSRs (except the last), e3 must access the HTTP server e5 on TCP port 80 and e4 must access the POP3 server e6 on TCP port 110, while all other traffic between these nodes must be blocked. Since the communication paths intersect at AP

p15 with e1 and e2, placing a firewall at p15 is enough to satisfy these constraints with minimal overhead. The final NSR requires that e4 cannot contact e3: this traffic does not go through p15 and no other shared path exists that can be filtered centrally. Thus an additional firewall must be allocated, either at p13 or p14, to block traffic from e4.
Performing this allocation operation manually by a designer would probably introduce configuration mistakes and an unoptimal configuration.
Instead, applying VEREFOO framework ensures formal correctness and optimal resource utilization. The final logical topology produced is the one present in Fig. 2.7, while Table 2.3 defines the resulting filtering policies of the allocated firewalls, including their default actions (denoted with a 'D').

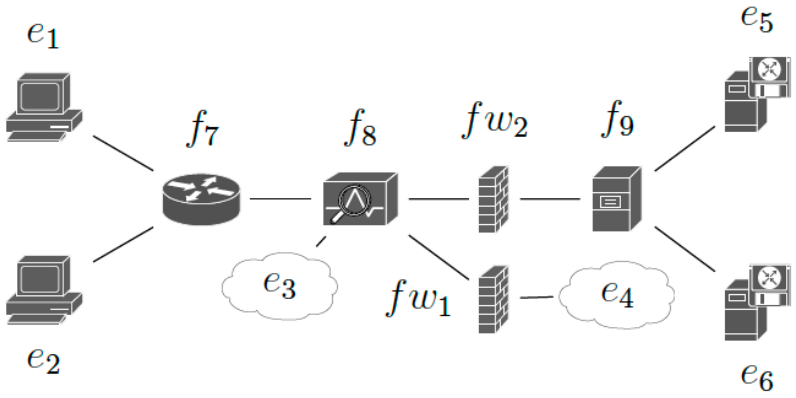

Figure 2.7. After: Service graph with allocated firewall

Firewall fw1						
#	**Action**	**IPSrc**	**IPDst**	**pSrc**	**pDst**	**tProto**
1	Allow	220.226.50.3	130.192.120.*	*	*	*
2	Allow	130.192.120.*	220.226.50.3	*	110	TCP
D	Deny	*.*.*.*	*.*.*.*	*	*	*
Firewall fw2						
#	**Action**	**IPSrc**	**IPDst**	**pSrc**	**pDst**	**tProto**
1	Allow	130.192.225.*	220.226.50.2	*	80	TCP
2	Allow	130.192.120.*	220.226.50.3	*	*	*
3	Allow	220.226.50.*	130.192.*.*	*	*	*
D	Deny	*.*.*.*	*.*.*.*	*	*	*

Table 2.3. Policy Rules

2.2.9 React-VEREFOO: an automated framework

So far, the software VEREFOO framework was explored, understanding how it works and its limitations, but its functionality can be extended to enable it as a fully autonomous component thanks to React-VEREFOO: it is designed to automate all the process of attack mitigation, from threat detection to policy deployment. This approach directly addresses the need for rapid response in dynamic network environments, where manual intervention is too slow and error-prone [21].
The major innovation of React-VEREFOO is its ability to use real-time data as input for the formal verification process, instead of relying on a static set of Network Security Requirements (NSRs), deriving new security requirements directly from logs generated by Intrusion Detection Systems (IDSs). When an IDS detects a suspicious activity, React-VEREFOO automatically formulates new NSRs to contain the threat. These new requirements are then sent into the MaxSMT solver, which re-evaluates the problem to generate an updated, formally correct and optimized firewall policy.
The updated configuration is automatically sent to the deployed firewalls, allowing in small time to reconfigure the network neutralizing the threat without human intervention. By integrating threat detection with policy enforcement, React-VEREFOO closes the loop on security automation, proving that formal verification can be a core component of a real-time, autonomous security orchestration system.
The Fig. 2.8 shows the approach of React-VEREFOO:

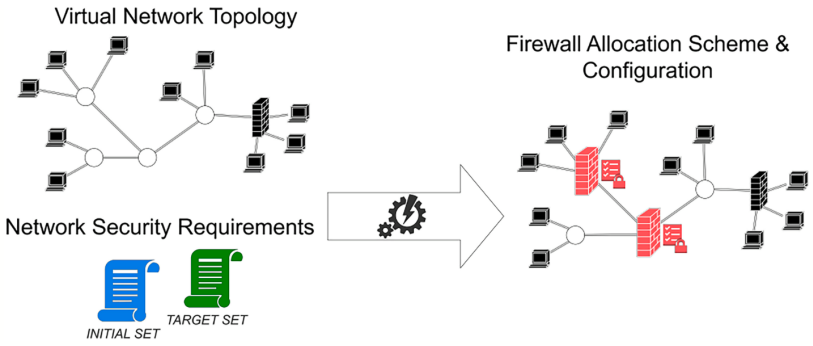

Figure 2.8. React-VEREFOO's approach

React-VEREFOO's steps

The major steps of React-VEREFOO are below described. The first step is to perform a high-level classification of all network security policies, comparing the initial set of policies with the new one, allowing to sort them into three groups:

- Added Policies: new policies that need to be enforced and were not part of the initial configuration.
- Deleted Policies: policies that were present in the initial configuration but are no longer needed in the target configuration.

- Kept Policies: policies that are already configured in the network and must remain in the final configuration.

The primary focus is on the "added" policies as they lead the reconfiguration process, so the algorithm should determine which network elements are in conflict with these new policies and must be modified. This process is handled in different ways, based on the new policy requirement:

- For a new isolation requirement (like a policy to block specific traffic), the algorithm examines all possible paths that the traffic could manage through the network, and, for each one, it checks if there is an existing firewall that is already blocking that traffic. If it finds a path where no firewall is blocking the traffic, it means that the current network configuration is not sufficient to enforce that new isolation policy: the algorithm marks all the nodes through that unblocked path as candidates for reconfiguration; then a solver, in a second moment, will decide where a new firewall should be placed or which existing one should be reconfigured to enforce the policy.

- For a new reachability requirement (like a policy to allow specific traffic), the logic is reversed: the algorithm scans all possible paths to see if at least one path is successfully unblocked from source to destination. If at least a path is found, the new policy is considered already satisfied and no reconfiguration is necessary. But, in the unfortunate case where all possible paths are blocked by a firewall, the algorithm identifies every firewall that is causing the blockage and adds these specific firewalls to the list of nodes that must be reconfigured. This ensures that the final solution will modify one (or more) of these blocking firewalls, thereby creating at least one valid path for the traffic and satisfying the new reachability requirement.

For a better understanding of the logic, the example present in [21] shows the addition of isolation requirement:

Algorithm 1 Algorithm for selecting network area to reconfigure for each added isolation requirement.

Input: an isolation requirement r, and an AG $\mathcal{G}_A$
Output: nodes to be reconfigured $N_{reconfigure}$

1: **for** $f \in F_r$ **do**
2: $found \leftarrow False$
3: **for** $n_i \in \pi(f) = [n_1, n_2, \dots, n_d]$ **do**
4: **if** $allocated(n_i)$ & $deny_{n_i}(\tau(f, n_i))$ **then**
5: $found \leftarrow True$
6: break
7: **end if**
8: **end for**
9: **if** $found == False$ **then**
10: $N_{reconfigure} \leftarrow \pi(f)$ ▷ All nodes in the path should be reconfigured
11: **end if**
12: **end for**
13: **return** $N_{reconfigure}$

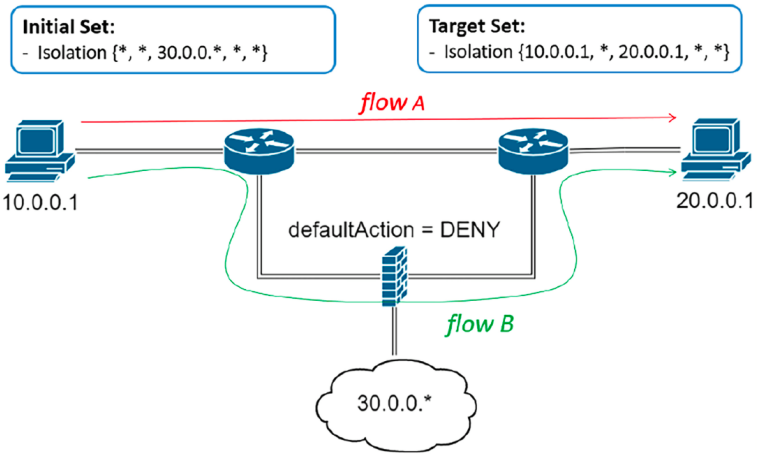

Figure 2.9. Example of adding an isolation requirement

2.2.10 Example of VEREFOO input structure

One of the most complete example to illustrate the structure required by VEREFOO for the input XML file is the *demo_input.xml* file. This file shows all the field and information required by the framework for a complete view of the network. The most important parts of the file are reported below.

```
<?xml version="1.0" encoding="UTF-8"?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
 <graphs>
  <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="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>
  <node functional_type="WEBCLIENT" name="40.40.42.-1">
   <neighbour name="1.0.0.6" />
   <configuration description="e5" name="businessofficeA">
    <webclient nameWebServer="130.10.0.1" />
   </configuration>
  </node>
    <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>
    <node name="1.0.0.9">
    <neighbour name="33.33.33.3" />
    <neighbour name="220.124.30.1" />
   </node>
   <node functional_type="NAT" name="220.124.30.1">
    <neighbour name="1.0.0.9" />
    <neighbour name="1.0.0.10" />
      <neighbour name="1.0.0.11" />
    <configuration description="s12" name="nat">
      <nat>
        <source>192.168.3.-1</source>
        <source>192.168.2.-1</source>
      </nat>
    </configuration>
   </node>
    .....
  </graph>
</graphs>
<Constraints>
 <NodeConstraints/>
 <LinkConstraints />
</Constraints>
<PropertyDefinition>
```

```
    <Property graph="0" name="ReachabilityProperty" src="40.40.41.-1"
            dst="130.10.0.4"  dst_port="80" lv4proto="TCP"/>
  <Property graph="0" name="IsolationProperty" src="40.40.41.-1"
            dst="130.10.0.1" dst_port="0-79" lv4proto="TCP" />
  <Property graph="0" name="ReachabilityProperty" src="40.40.42.-1"
            dst="40.40.41.-1" />
    <Property graph="0" name="IsolationProperty" src="40.40.41.-1"
            dst="40.40.42.-1" />
    .....
  </PropertyDefinition>
</NFV>
```

For a more in-depth analysis of VEREFOO's implementation, you can consult the project's official GitHub page.

Chapter 3

Thesis Objectives

Modern computer networks are characterized by increasing complexity and evolving cyber threats, so ensuring an effective security management in this environments requires automated solutions. However, a major obstacle lies in the lack of compatibility between the data formats used for network monitoring and the rigid input requirements of formal verification tools. In particular, these tools, like VEREFOO, require an input in XML format and in a precise structured way, while operational data is usually collected in JSON, following the OpenC2 standard. This strictness is necessary because, as demonstrated in [18], VEREFOO uses a sophisticated modeling approach on packet classes to achieve high scalability in complex virtualized environments. This main difference requires security analysts to manually translate this data, which is time-consuming and very prone to errors.
The main objective of this thesis is the development of a **connector** that can automatically take network information from several OpenC2 JSON files, each one of them describing a different aspect of the network, and merging them into a single XML file suited for VEREFOO, to allow to easily understand and check for the network's security rules. This automated conversion process eliminates the manual task of data translation but also ensures that the network security can be continuously validated in a scalable, reliable manner, which is essential for managing large-scale, dynamic network infrastructures [22].

3.1 Connector Architecture and Functionality

To achieve this goal, the development focuses on several technical challenges. The connector, implemented in Python, is structured into logical phases, with each phase addressing a specific aspect of the data transformation process.

3.1.1 OpenC2 JSON Data Analysis

The process begins with the analysis and extraction of network data from the input JSON files. The connector is built to navigate the nested structure of OpenC2 Response messages, focusing on the `x-ctxd` (OpenC2 Context Discovery) field to extract the details from the services and links sections. A dedicated function, `get_nested_value`, ensures the correct access to the data, preventing any errors. The `deduce_service_role` function is designed to gather the role of a network element (for example FORWARDER, WEBCLIENT) analyzing key attributes like protocol, local name and description. With this approach, the connector extracts valuable data and also performs model interpretation, which is crucial for VEREFOO's formal verification. The implementation of `deduce_service_role` in Python utilizes regular expressions (`re` library) to perform keyword matching in a case-insensitive way, making the detection process more robust and ensuring that the service roles are accurately inferred from various textual descriptions within the JSON data.

3.1.2 Data consolidation and Normalization

After the extraction of the data for all the input files, the connector consolidate them into a single network model, processing each link service and link entry, mapping logical names and hostnames to the corresponding IPv4 address, ensuring that the final output accurately represents the network. The use of dictionaries and sets is the key factor to prevent duplicate entries and maintain data integrity. This process covers not only the aggregation of different JSON sources, but also the normalization of the various identifiers, ensuring that the output file represents the network graph in an accurate and non redundant way. Specifically, the connector uses dictionaries to store network elements, with their names or IP addresses serving as unique keys to prevent duplicate entries. This approach guarantees that even if multiple JSON files describe the same network component, it will be represented only once in the final model, thus creating a clean and consistent graph, with all the information retrieved by the different file for each node.

3.1.3 VEREFOO XML Structure Generation

This phase involves the construction of the XML output file according to VEREFOO's strict schema. The connector uses Python's `xml.etree.ElementTree` library to build the XML tree element, from the root <NFV> tag with its schema location (`nfvSchema.xsd`) to the sub-elements like <graphs>, <graph>, <nodes> and <node>. The correct functional type (like WEBCLIENT, FORWARDER) and attributes are automatically assigned to each node, ensuring complete compliance with VEREFOO's input requirements. The connector also automatically adds the necessary tags, such as <Constraints> and <PropertyDefinition>, which are required by VEREFOO for validation, but their definition is not part of this thesis objective.
In addition, the created XML tree undergoes an additional pass through a `prettify` utility that adds spaces and indents, ensuring not only the required formal validity of the output, also its readability and utility.

The generation of the XML structure is a key function of the connector: in fact, after processing the JSON data, the connector iterates through the consolidated `service_map` to create each <node> element. For each service, it sets the `functional_type` attribute based on the `deduced role` and assigns the `name` attribute using the service's IP address. It also populates the <neighbour> tags by referencing the `link_map` to build the graph topology. The use of the `xml.dom.minidom` library then processes the raw XML tree, applying indentation and line breaks to produce the final file. Additionally the generated XML file is merged with a specified file containing the security properties, ensuring stability and execution speed.

3.1.4 Connector Management and Graph Coherence

This section is quite technically sophisticated because the key challenge in generating network graphs is ensuring the consistency and bidirectionality of connections. The connector automatically checks and enforces it: if a node A lists node B as its neighbour, the code automatically checks that node B contains A as a neighbour as well, adding the missing link if necessary. Additionally, the connector creates new nodes for network elements that are only referenced as neighbour in the input data but not explicitly defined as services, further ensuring that the final graph is as complete as possible. This consistency check is implemented by iterating through all discovered links and for each link from node A to node B: the code explicitly verifies if a reciprocal link exists from B to A in the `link_map`. If the link is unidirectional in the raw data, the connector adds the reverse link to guarantee bidirectionality.

Benefits and Impact of this Solution

The use of this automated connector provides significant benefits for the field of network security:

- *Automation and Error Reduction*: just eliminating the manual data translation, the connector enormously reduces the potential for human error and ensures that the network model is always accurate and consistent.
- *Increased Efficiency*: the entire conversion process, which could take hours of manual effort and high skills, is completed in seconds, allowing security personnel to focus on higher-level analysis and threat mitigation.
- *Seamless Integration and Workflow*: the automated merge of the generated XML with the user-provided security properties file enables a single, complete execution. This eliminates the need for a manual, intermediate step, making the entire workflow smoother, more reliable, and significantly faster.
- *Continuous Security Posture Management*: this solution allows continuous security checks, as can be integrated into automated workflows, enabling organizations to run VEREFOO analyses on every minor network change to ensure security rules remain optimized and without any misconfigurations.

In summary, this thesis provides a trusted practical tool to address the needs in network security automation, making easier and quicker to manage complex systems in a more reliable and automated way, avoiding manual mistakes. Automated and verifiable solutions are extremely important in dynamic environments such as SDN-based IoT systems, as discussed in [23], where the emphasis is placed on policy-driven approaches to security orchestration.

3.1.5 Submitting the XML to VEREFOO for Formal Verification

The second phase of the data transformation process involves submitting the final XML file to VEREFOO. The connector uses a RESTful API to interact with VEREFOO, targeting a specific endpoint that initiates a simulation. The process begins with an HTTP POST request, sending the XML file content as the request body. This action triggers VEREFOO to perform its analysis on the provided network model. The VEREFOO API is designed to handle this process either synchronously or asynchronously. If the analysis is performed in real-time, the response contains the complete results. However, for more complex network models, which VEREFOO is optimized to handle [18], the API returns a status response with a URL to a resource where the results will be available once the analysis is finished. To handle this, the connector implements a robust polling mechanism, making repeated HTTP GET requests to the provided URL with an exponential backoff strategy, ensuring that the final verification output is successfully retrieved. This automated submission and retrieval process eliminates the need for manual intervention, making the entire workflow extremely efficient for security analysts. The Python `requests` library is used to perform the HTTP POST to the specific VEREFOO endpoint, which is hard-coded as /verefoo/adp/simulations.

3.1.6 Parsing the VEREFOO XML Response and Generating OpenC2 Commands

Upon receiving the full XML response from VEREFOO, the connector acts as a parser to extract actionable information. The core of this process is to identify the security properties that have been violated during the formal verification. The VEREFOO output includes a propertyDefinition section, where each property, such as `REACHABILITY_PROPERTY` or `ISOLATION_PROPERTY`, is detailed along with its satisfaction status (isSat). Using Python's `xml.etree.ElementTree` library, the connector navigates this structure to find properties where the isSat tag has a value of false. For each violation, the parser extracts key attributes of the property, including source (src), destination (dst), protocol (protocol) and ports (src_port, dst_port). This extracted data is then used to construct structured OpenC2 commands. This process is part of the larger autonomous security framework, allowing the system to automatically generate and deploy new security policies to mitigate detected threats, as described in [22]. Specifically, a violated REACHABILITY_PROPERTY triggers the generation of an allow command, while a violated ISOLATION_PROPERTY leads to a deny command. The commands are formatted

into a JSON-like structure that can be easily consumed by an OpenC2 actuator, thereby enabling the automatic reconfiguration of network devices (like firewalls) to enforce the required security policies.
The connector for this parsing and generation is explicit: it checks the isSat attribute of each property. If the status is true, it builds a string representing the OpenC2 command. For example, for an isolation property violation, it generates a line of code like cmd = oc2.Command(oc2.Actions.deny, oc2.IPv4Connection(src_addr='...', dst_addr='...'), actuator=pf), which can then be saved to a file for direct execution by the orchestration layer.

Chapter 4

Implementation

The implementation phase of this thesis focuses on the development of a connector that bridges the gap between OpenC2 Context Discovery (CTXD) responses and VEREFOO's XML required input format. The connector not only parses the network context into VEREFOO's XML but also automatically merges the generated XML with a separate security properties file, submits the XML to the VEREFOO API for formal verification, parses the returned results, and, if the outcome is satisfiable, generates the file of actionable OpenC2 commands to enforce the required security policies. This chapter details the overall system architecture, the design choices made for data model mapping and the specific aspects of the logic.

4.1 System Architecture and Workflow

The connector is implemented as a single Python script (connector.py) that functions as an automated parser. The workflow is as follows:

- Input Acquisition: the script takes a directory, passed as command-line input and using the `argparse` module to handle it, containing the OpenC2 CTXD responses stored as multiple JSON files. This initial step aggregates network data from different sources into a unified model.

- Data Extraction: the script iterates through each JSON file, parsing its content to extract relevant network context data, including services and links. The `extract_data_from_json`, `extract_service_data` and `extract_link_data` functions are part of this step.

- Data Aggregation: the extracted data from all input files is collected into a single comprehensive dataset. This aggregated data provides a holistic view of the entire network topology, achieved by appending the data from each processed JSON file into a list named `all_extracted_data`, which is then passed to the XML generation function.

- XML Generation: the core function of the connector, the `create_xml_from_data` function, processes the aggregated data to build the VEREFOO-compliant XML tree. This involves a multi-pass approach to ensure all network elements are present and correctly represented, including the automatic assignment of IP addresses and functional roles. The final XML file is the formal representation of the network topology, ready for VEREFOO's verification.
- Security Property Integration: the connector then proceeds to consolidate the newly generated XML with a separate file containing the security properties. This step creates a single and comprehensive input, eliminating the need for manual file combination and streamlining the entire workflow.
- VEREFOO API Interaction: the connector uses the `requests` library to send the generated XML to the VEREFOO REST API. It manages the full API communication, including handling the response. This step is where the formal verification process begins.
- Output Parsing and Command Generation: once VEREFOO returns its XML response, the connector acts as a final parser. It reads the VEREFOO output, specifically looking for `isSat="true"` statuses on properties like 'REACHABILITY_PROPERTY' or 'ISOLATION_PROPERTY'. Based on the specific policy violation, the script generates a corresponding OpenC2 command ('allow' or 'deny'), translating the abstract verification result into a concrete security action.
- Actionable Output: the generated OpenC2 commands, formatted as Python code strings, are saved in a file. This file can then be used by an OpenC2 actuator (such as an iptables or firewall manager) to automatically implement the necessary security changes, completing the security management loop.

A graphic visualization of the main phases connector's workflow is reported in Fig. 4.1:

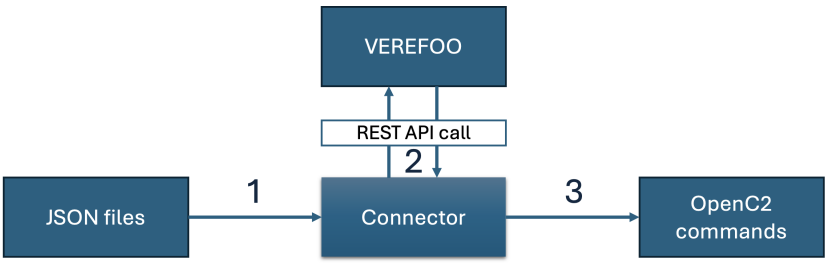

Figure 4.1. Connector workflow

4.2 Data Model Mapping and Design Choices

A key challenge addressed by the connector is the mapping between the hierarchical data model of OpenC2's CTXD JSON responses and the rigid NFV schema required by

VEREFOO (XML). The main goal of this mapping is to store the information in an accurate, complete and coherent way for the formal verification requirements.
The mapping involves:

- *Node identification and naming*: VEREFOO requires a unique identifier for each node in its graph. The connector is designed to prioritize the most concrete network identifier available in a specified hierarchy, being set to a service's IPv4 address, when found, to provide a verifiable network identity. In cases where an IPv4 is not available, the connector falls back to the other unique identifiers like the service's hostname or logical name. The choice to prioritize IPv4 is strategic because it is a fixed and universally recognized network address, making it the most reliable identifier for VEREFOO's graph-based analysis. A temporary IPv4 address is assigned to nodes lacking a definitive IP, ensuring every node in the VEREFOO graph has a verifiable network address. The code uses an arbitrary predefined IP range, specifically 20.0.0.1 and increments it for each new node, to prevent clashes with existing network IPs.

- *Automated functional_role deduction*: the `deduce_service_role` function is implemented to automatically assign a `functional_type` (for example NAT, FORWARDER, WEBCLIENT). This function uses heuristics based on protocol information, keywords in the service's name or description (e.g., nat, database), and link types (protect) to deduce the node's role. This is crucial for VEREFOO to apply the correct verification logic for the nodes without all the information required.

- *Exhaustive topology synthesis*: it is a two-pass approach for building the network topology. Firstly it processes all the explicitly defined services and then performs another pass to identify and create nodes for network elements that are only mentioned as peers within the links section of the JSON data. This ensures that the generated XML graph is a complete representation of the network, including implicit or "neighbour-only" nodes. The `create_xml_from_data` function creates the nodes for all services found in the JSON responses; then, in a second loop, it checks for any neighbour names that do not correspond to an existing node and creates a new node with a deduced functional type for them.

- *Data deduplication*: `service_map` tracks all processed nodes to prevent the creation of duplicate <node> elements. Similarly, a `seen_neighbour_ipv4s` set is used to ensure that a node's neighbours are listed only once, even if they are referenced multiple times in the JSON links. This is used for preventing redundant entries and maintaining an accurate graph for the VEREFOO solver.

4.3 Connector Logic and Implementation Details

The connector is implemented as a Python script employing standard libraries for JSON parsing and XML manipulation. The core logic is developed in different key functions:

- `extract_service_data(svc)`: processes individual OpenC2 service objects, extracting their name, ID, hostname, associated links and description. An important

aspect is how it employs the `deduce_service_role` function to assign an heuristically deduced `functional_type` (for example FORWARDER, WEBCLIENT) and to handle specific configurations like NAT rules. The function begins by initializing a dictionary `service_data` with default values for various fields. Subsequently it iterates through the service's type dictionary, which can contain multiple entries, to extract the id, hostname, and description, giving a more complete overview of the service. It calls `deduce_service_role` to automatically determine the service's function, ensuring that, if a specific role like "NAT" is identified, it specifically looks for `nat_rules` in the configuration section to extract the `nat_sources` for later XML generation.

- `extract_link_data(link)`: parses OpenC2 link objects, extracting their name, description, type and detailed peer information, including service names and IPv4 addresses. It handles cases where service names might include embedded IPv4s. The function creates a `link_data` dictionary and processes each peer within the link. A key part of its logic is the handling of peer names that may contain a newline character (`\n`) separating a logical name from an IPv4 address, which is a specific format found in different OpenC2 responses. This logic splits the string to correctly capture both the logical name and the IPv4 address, which is critical for accurate node naming in the VEREFOO XML output.

- `extract_data_from_json(json_data)`: takes a complete OpenC2 JSON response, navigates to the `x-ctxd` section and calls the `extract_service_data` and `extract_link_data` functions to gather all relevant services and links. The function first checks if the `x-ctxd` section exists, as it is the root of all the contextual information. If it is missing, the function returns an empty dictionary, preventing crashes. It then uses list comprehensions to process all services and links, applying the extraction functions to each element and collecting the results. This approach ensures that all relevant data from the JSON is captured in a structured format.

- `create_xml_from_data(all_extracted_data)`: it is the main function that effectively build the VEREFOO XML. It iterates through all extracted services, creating a <node> for each. For each node, the connector first ensures that it has a valid name attribute, prioritizing the IPv4 address if available, and assigns the `functional_type`. After resolving any missing IPs with fallback arbitrary addresses and consolidating neighbour references, the connector generates the <configuration> block with all relevant details (ID, hostname, IPv4, link type) and then adds <neighbour> elements based on the established adjacency relationships. The function includes a multi-pass approach to handle nodes without explicit IPs: it identifies these nodes, assigns them a unique arbitrary IPv4 from a reserved range (20.0.0.1) and then renames all neighbour references accordingly to ensure a fully consistent network graph. For each node with additional information, it generates the <configuration> block, including detailed information like NAT sources and then adds <neighbour> elements based on the discovered links. An important step is the handling of duplicate neighbour IPv4s. A `seen_neighbour_ipv4s` set is used for each node: before adding a new <neighbour> element, the script checks if the corresponding IPv4 is

already present. This prevents redundant neighbour entries for a single node, which can arise from the structure of OpenC2 links, ensuring that VEREFOO's graph model is clean and accurate.

4.4 Dataset and Examples

The input for the connector is derived from OpenC2 Context Discovery responses, provided in JSON format. For this thesis, a representative set of 21 example input files serves as the primary dataset (described in details as Example 0 in subsection 5.2.1). These files are structured in the typical network context structure that an OpenC2 Consumer provides, including details about services (such as containers, virtual machines, nat, cloud, etc) and their interconnections (links). The JSON structure provides granular information like service id, hostname, type, owner and actuator details.
The successful execution of the connector generates a VEREFOO XML file, which represents the XML file that adheres to VEREFOO's NFV schema, formalizing the network topology and service configurations for verification. The process is a direct mapping from the JSON-structured output of OpenC2's Context Discovery to the VEREFOO XML format. A service, described as a container in the JSON input file, with its unique identifier, hostname, and associated IPv4 address, is systematically transformed into a <node> element. This <node> element is populated with key attributes: a `name` attribute that prioritizes the IPv4 address as a unique network identifier and a `functional_type` (like WEBCLIENT, FORWARDER) returned from the `deduce_service_role` function. Nested within the <node> are the <neighbour> block and <configuration>, if other information is present.
An example that shows the complete structure of the extracted information parsed for the XML file is the "vm" (Virtual Machine) node with IPv4 addresses "192.168.0.201". An important aspect is the presence on neighbours which, based on the input file provided, are present only in specific type of nodes: this issue is discussed further along.

```
<node functional_type="WEBCLIENT" name="192.168.0.201">
    <neighbour name="10.17.1.45"/>
    <neighbour name="10.17.1.46"/>
    <neighbour name="10.17.1.47"/>
    <neighbour name="10.17.1.48"/>
    <neighbour name="10.17.1.49"/>
    <neighbour name="10.17.1.50"/>
    <neighbour name="20.0.0.2"/>
    <neighbour name="20.0.0.3"/>
    <neighbour name="20.0.0.4"/>
```

As described before, the 20.0.0.4 ("os-fw") node is not explicitly defined in services but it is present as neighbour. When the connector identifies a node name in a neighbour list that is not defined, it proceeds to create a new node entry for it. To the absence of detailed service information, due to the absence of an explicit definition, the connector applies a heuristic approach to deduce the node's `functional_type` based on its name (for example

if the name contains nat, the deduced role will be NAT), while for the firewall node the connector set the functional_type as None, allowing a more flexible use from VEREFOO. This is a critical design choice because it ensures that even partially defined networks can be fully represented and verified, preventing gaps in the topological graph that could lead to verification failures. The `create_xml_from_data` function specifically includes a loop that scans all neighbour elements of the already created nodes: if a neighbour's name (which could be an IP or a logical name) does not match any existing node's name, the script dynamically creates a new node element for it and uses the `deduce_service_role` function to assign a `functional_type` based on keywords in the name.
This is another important aspect that needs to be highlighted: the connector maintains a correct network representation, guaranteeing the bidirectionality of neighbour relationships. For VEREFOO to perform a complete and accurate formal verification, the network topology must be fully connected and consistent. If a node (Node A) lists another node (Node B) as a neighbour, the connector must ensure that Node B also includes Node A in its list of neighbours, even if not directly defined. The `create_xml_from_data` function manages this by firstly building an adjacency list-like data structure called `neigh_list_after_ip_assignment`, which maps each node to a set of its neighbours. It then iterates through this map and, for every neighbour-to-neighbour relationship found (A is a neighbour of B), it ensures that the reverse relationship (B is a neighbour of A) is also added to the set. Finally, it rebuilds the XML tree by removing all original <neighbour> elements and creating new ones based on this symmetrical, verified adjacency list. This ensures that the final XML graph is undirected and consistent, satisfying a key requirement for VEREFOO. This fixed approach can be seen in "os-fw" XML output:

```
<node name="20.0.0.4">
    <neighbour name="192.168.0.200"/>
    <neighbour name="192.168.0.201"/>
    <neighbour name="192.168.0.202"/>
    <neighbour name="20.0.0.2"/>
</node>
```

4.5 Connector XML Output

The output of the connector is an XML file structured according to VEREFOO's NFV schema, which serves as the formal representation of the network graph and its components. The XML structure begins with the root <NFV> element, which includes the schema definitions. Nested within there are <graphs> containing a single <graph id="0">, which cover the <nodes> section. The <nodes> section lists all the network elements discovered. Below the graphs, the XML defines a <PropertyDefinition> section, which is required by VEREFOO for specifying verification properties, though they remain empty as they are beyond the scope of this thesis.
Each network element discovered by OpenC2 CTXD is represented as a <node> within the XML. Key attributes of the <node> element include:

- `name`: essential for identifying the node within VEREFOO's graph. It is dynamically

set to the service's IPv4 address if available, providing a concrete network identifier.

- `functional_type`: derived by the `deduce_service_role` function, specifies the role of the node (for example WEBCLIENT, NAT). This is essential for VEREFOO to apply the correct verification logic and allocate the appropriate security functions. For instance, a node identified as `functional_type="WEBCLIENT"` specifies its role.

Immediately after the `functional_type` attribute, specific type of <node> contains a <configuration> element. This block provides additional detailed information that VEREFOO can leverage for its verification and optimization. A specific improvement is the specific handling of configurations for NAT nodes, where a <nat> block is created with <source> tags for each source IP address. This detail is needed for VEREFOO to understand how the NAT is configured and to correctly model network traffic. The configuration details are stored in a `config_details_map` and then used to populate the XML tree, ensuring that only relevant configuration blocks (<nat>, etc.) are generated for the correct functional types. After the <configuration> section, the <node> element includes one or more <neighbour> tags which formally represent the direct connectivity between nodes in the network graph. Each <neighbour> element has a name attribute, which is set to the IPv4 address of the neighbouring node if present. The de-duplication logic (using `seen_neighbour_ipv4s`) ensures that each unique IPv4 neighbour is listed only once, preventing duplicates and maintaining an accurate network topology for VEREFOO. This ensures the integrity and correctness of the network graph, which is vital for the formal verification process to properly function and produce reliable results.
This structured XML output accurately translates the network context into a format that VEREFOO can elaborate, using the formal verification of security policies on the discovered network topology.

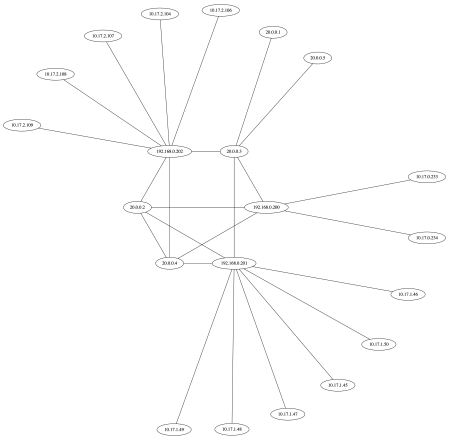

Figure 4.2. Network topology

For the graphs in this thesis the Graphviz Online tool has been used, specifying the DOT language for the visualization of the network graph generated by the connector.

4.6 Submitting the XML to VEREFOO

After the generation of the XML file, the script proceeds with the interaction with the VEREFOO framework and the subsequent generation of OpenC2 commands:

- Automated Input Merging: the connector automatically merge the generated XML network topology with a separate file containing security properties. This is achieved by concatenating the contents of both xml files into a new XML structure that includes both the <graphs> and <propertyDefinition> sections. The merged XML is saved to a temporary file, which is then used for the VEREFOO API submission. In this manner, it creates a single and complete input file for VEREFOO without requiring any manual intervention.
- VEREFOO API Interaction: the script initiates a POST request to the VEREFOO REST API endpoint, by default verefoo/adp/simulations, sending the complete XML file (which must include <propertyDefinition> manually added by the user for security policies). It sets the headers to handle XML content while VEREFOO processes this input for formal verification and optimization. A key feature of the connector is its handling of API responses. The `requests.post` call might not immediately return the full result. The script checks for a nested <content> element in the response to determine if the full XML output is present. If not, it looks for a <href> link within a <links> element, which points to the location of the final result. The script then automatically makes a subsequent GET request to this URL, effectively managing VEREFOO's asynchronous job execution to retrieve the complete verification output.
- VEREFOO Output Processing: the response from VEREFOO, an XML file containing the verification results (SAT/UNSAT) and optimized firewall policies (if applicable), is saved. Crucially, the script checks that the VEREFOO output is not "UNSAT": if the security policy cannot be satisfied (UNSAT), the process stops, and no OpenC2 commands are generated. This prevents the deployment of ineffective or impossible policies. The script uses the `ET.fromstring` function to parse the XML response from VEREFOO. It then performs a case-insensitive check for the string "UNSAT" within the XML content. This ensures, that regardless of how VEREFOO formats the unsatisfied result, the connector recognizes it and stops the process. This check prevents the connector from generating commands for policies that are provably unreachable or violated.
- OpenC2 Command Generation: if VEREFOO's simulation result is "SAT", the script proceeds to parse the VEREFOO output XML file. It extracts information about verified security properties (ReachabilityProperty and IsolationProperty), along with source/destination IPs, protocols, and ports if present. Based on whether a property is "REACHABILITY_PROPERTY" or "ISOLATION_PROPERTY" and its satisfaction status ("true"), the script constructs corresponding OpenC2 allow or deny commands. These commands are then written to a specified output file, formatted for an OpenC2 consumer, such as iptables rules. This final step translates the formally verified policies into executable network security actions.

An example that shows the VEREFOO's output is the following:

```
<node>
 <neighbour>
  <neighbour>
   <id>131</id>
   <name>10.17.1.45</name>
  </neighbour>
  <neighbour>
        <id>132</id>
   <name>10.17.1.46</name>
  </neighbour>
  <neighbour>
   <id>133</id>
   <name>10.17.1.47</name>
  </neighbour>
  <neighbour>
   <id>134</id>
   <name>10.17.1.48</name>
  </neighbour>
  <neighbour>
   <id>135</id>
   <name>10.17.1.49</name>
  </neighbour>
  <neighbour>
   <id>136</id>
   <name>10.17.1.50</name>
  </neighbour>
  <neighbour>
   <id>137</id>
   <name>20.0.0.1</name>
  </neighbour>
  <neighbour>
   <id>138</id>
   <name>20.0.0.3</name>
  </neighbour>
  <neighbour>
   <id>139</id>
   <name>20.0.0.1</name>
  </neighbour>
 </neighbour>
 <configuration/>
 <id>17</id>
 <name>192.168.0.201</name>
 <functionalType>WEBCLIENT</functionalType>
```

```
</node>
...
<propertyDefinition>
 <property>
  <property>
   <pop3Definition/>
   <name>REACHABILITY_PROPERTY</name>
   <graph>0</graph>
   <src>192.168.0.201</src>
   <dst>10.17.1.50</dst>
   <lv4Proto>ANY</lv4Proto>
   <srcPort/>
   <dstPort/>
   <isSat>true</isSat>
   <body/>
   <httpdefinition/>
  </property>
    ...
    </property>
</propertyDefinition>
```

4.7 Parsing the VEREFOO Response and Generating OpenC2 Commands

The final and most crucial step of the pipeline is the translation of VEREFOO's formal verification results into actionable OpenC2 commands. The connector acts as a parser, with its primary focus on the isSat field, which is nested within each <property> element of the VEREFOO response: the script only generates an OpenC2 command file when a property is satisfied (isSat=true). An UNSAT value signifies that the policy cannot be fulfilled, so the script alerts the user and stops the command generation process, preventing the deployment of invalid security rules. If the value is true, the connector continues with the parsing and command generation.

- REACHABILITY_PROPERTY: when VEREFOO reports a true result for a reachability property, it means that the required communication path is successfully verified and can be enabled. Consequently, it generates a Python-style OpenC2 command with the allow action, aiming to ensure the connection remains open. The script specifically looks for the relevant <property> elements in the parsed XML tree, then extracts the name, src, and dst fields, which define the details of the policy. The protocol, src_port, and dst_port are also extracted if available, ensuring the generated command is as specific and detailed as possible, creating precise and non-ambiguous OpenC2 commands.
- ISOLATION_PROPERTY: a true result for an isolation property is a direct indication of a security success. It means that a connection that should be blocked

is verifiably blocked. To enforce it, the connector generates an OpenC2 command with the deny action. This command is designed to prevent unauthorized traffic, re-establishing the network's security and satisfying the policy. The logic for generating a deny command is similar to the allow one, but it's triggered by a different property name. The script constructs the `oc2.Command` string by dynamically concatenating all the extracted connection arguments (e.g., src_addr, dst_addr, protocol). This dynamic construction makes the script very flexible, as it can adapt to various policy definitions from VEREFOO without needing a predefined command structure. The generated command strings are then written to the output file creating an executable script for an OpenC2 actuator.

For each satisfied property, the parser meticulously extracts all necessary details from the VEREFOO response, including source and destination IPv4 addresses, protocols (such as TCP, UDP, ICMP), and port numbers. These details are used to construct a precise and complete OpenC2 command, formatted as a Python code string (`oc2.Command`). This command is then saved to a file, making it ready to be consumed by an OpenC2 actuator, such as an iptables or a firewall manager. This automated step closes the security management loop, transforming the abstract data from network discovery and formal verification into a proactive and enforceable set of commands.
An example that shows the output file in OpenC2 is the following:

```
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
 src_addr=oc2.IPv4Net("192.168.0.201"),
 dst_addr=oc2.IPv4Net("10.17.1.50")),
 arg, actuator=pf)
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
 src_addr=oc2.IPv4Net("192.168.0.202"),
 dst_addr=oc2.IPv4Net("10.17.2.104")),
 arg, actuator=pf)
...
cmd=oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
 src_addr=oc2.IPv4Net("10.17.1.45"),
 dst_addr=oc2.IPv4Net("192.168.0.201")),
 arg, actuator=pf)
```

4.7.1 File I/O and JSON Parsing

The initial step of input acquisition involves reading and parsing JSON files. The script uses a `try...except` block to wrap the file reading and JSON parsing operations. Specifically, if a file at the specified input path does not exist (FileNotFoundError), or if a file is found but its content is not a valid JSON object (json.JSONDecodeError), the script will catch the exception, print the error message to the console, and continue processing other files in the directory without crashing the entire workflow. In addition, every input file is processed independently, ensuring that a single corrupted file cannot stop the rest of the workflow. Nested key lookups within JSON structures are always checked by a

helper function that falls back to safe default values, thus avoiding KeyError or TypeError exceptions when expected fields are missing.

4.7.2 VEREFOO API Interaction Failures

The interaction with the VEREFOO REST API is a potential point of failure due to network issues, server unavailability, or incorrect request formats. To ensure the correct communication, the script encapsulates the `requests.post` and `requests.get` calls within another `try...except requests.exceptions.RequestException as e:` block. This allows the script to manage various network-related problems, such as connection timeouts, DNS resolution failures, or HTTP errors. Upon catching an exception, the script logs the specific error message, providing the user with immediate feedback on why the API call failed. It also prevents the script from proceeding to the following steps, which rely on a successful API response, thus avoiding subsequent errors. Furthermore, the script checks the presence of the selected XML file before attempting transmission, cleans problematic characters (such as non-breaking spaces \xa0), and validates whether the response contains either inline verification results or an external result URL before proceeding.

4.7.3 XML Structure and Parsing Errors

Once the VEREFOO API returns an XML response, the script must parse it to extract verification results. The `xml.etree.ElementTree` library is used for this purpose. To prevent the script from crashing if the returned XML is malformed or has an unexpected structure, the `ET.fromstring()` call is wrapped in a `try...except ET.ParseError as e:` block. This is especially important as external APIs can sometimes return corrupted data. Additional safeguards ensure that property definitions are skipped if incomplete or malformed, avoiding the generation of invalid commands.

4.7.4 Handling UNSAT Results from VEREFOO

The script explicitly checks VEREFOO's verification results for unsatisfiable conditions ("UNSAT"). If any UNSAT result is detected, the script stops before the generation of OpenC2 commands, preventing the execution of potentially invalid commands. This introduces an additional layer of robustness by coupling API feedback with execution control.

4.7.5 Default Value Assignment and Data Consistency

When generating the internal XML representation, the connector assigns fallback IP addresses to nodes lacking valid IPs and ensures that neighbour relationships are consistent. This prevents malformed XML or broken references that could cause later failures in command generation or external API interactions. To achieve this, the script validates IPv4 addresses against a clear pattern, guarantees uniqueness of assigned fallback IPs, and enforces bidirectional neighbour relationships (if A is neighbour of B, B becomes neighbour

of A). Duplicate neighbours are removed, and unknown service roles fall back to a default type (None).

4.7.6 Configuration and Command Generation Safeguards

Configuration tags within the XML (such as nat, cache, dpi, mailserver) are only generated if valid attributes are available, ensuring that incomplete or unnecessary elements are not created. During OpenC2 command generation, the script validates that all mandatory fields (name, src, dst, isSat) are present, checks the protocol against a predefined map, and only adds port arguments if values are properly defined. If no valid commands can be generated, the script explicitly notifies the user.

4.7.7 Global Exception Handling

Finally, the main execution flow is wrapped in a global exception handling so any unexpected errors poped up during the execution process are caught and reported to the user with descriptive messages, rather than causing abrupt termination of the program.

Chapter 5

Validation

The validation phase is essential to confirm that the implemented connector effectively converts OpenC2 Context Discovery (CTXD) JSON responses into a VEREFOO-compatible XML format so it can be successfully processed by VEREFOO for formal verification. This chapter outlines the methodology applied for the validation process, establishing a clear and repeatable framework for evaluating the connector's performance. Each example tests the connector against different network topologies and data integrity challenges and it presents the specific results obtained from the VEREFOO processing phase, demonstrating the accuracy and robustness of the connector in different formed experiments.

5.1 Validation Methodology

The validation process was designed to rigorously test the connector's functionality and its integration with the VEREFOO framework. The methodology follows a multi-step approach, simulating a complete network security orchestration workflow.

1. Input Preparation: the process begins by populating a dedicated directory with a set of representative OpenC2 CTXD JSON files. The dataset used for validation consisted of different files with a network elements without explicitly defined IPv4 addresses for testing purposes. These files describe the network components, including web servers, web clients, firewall, forwarder, along with their interconnections. The inclusion of files with incomplete data, such as missing IP addresses, was a clear choice to stress-test the connector's data-handling and role-deduction logic, simulating realistic conditions where network context might be fragmented or incomplete.

2. Data Transformation: the connector script is executed. The script processes the entire directory of JSON files, performs data aggregation and normalization, and generates a complete XML file that follows to the VEREFOO NFV schema. This step validates the connector's core functionality: the correct deduction of node roles, the dynamic assignment of unique identifiers (prioritizing IPv4) and the enforcement of graph coherence through bidirectional link validation. The multi-pass data

transformation process ensures that every piece of information is processed and cross-referenced. The first pass aggregates all discovered nodes, while a second pass is used to validate and create links between them, ensuring the generation of a complete and valid network graph. This two-phase approach is fundamental to maintain the graph integrity required by VEREFOO.

3. Security Properties Definition: following the generation of the base XML file, the connector automatically merges it with a user-defined file containing the security properties. This is done by populating the PropertyDefinition section of the generated XML file with the relevant security policies, such as ReachabilityProperty and IsolationProperty. These properties are crucial for VEREFOO execution, guiding the formal verification process, ensuring that the network model is evaluated against the specified security requirements.

4. Formal Verification with VEREFOO: the generated XML file is then submitted to the VEREFOO RESTful API for formal verification. The process confirms that the network topology and properties are correctly interpreted by the formal verification engine.

5. Output Analysis: the final VEREFOO XML response, which includes the satisfaction status (‘isSat‘) for each security property, is parsed by the connector. The script is designed to identify all policy violations (where ‘isSat="true"‘) and automatically generate corresponding OpenC2 commands (like allow for violated ‘ReachabilityProperty‘ and deny for violated ‘IsolationProperty‘).

5.2 VEREFOO Processing and Results

5.2.1 Example 0: Initial case

This example, which was used to create the connector script and is cited in the previous chapter, is fully described below.
This example demonstrates the connector script's ability to handle a complex and highly interconnected network topology with numerous nodes. The graph consists of multiple web clients grouped around several central nodes, which act as points of connection to the wider network. The topology includes WEBCLIENT, FORWARDER and other functional types, representing a realistic scenario with different network services.
This test case represents the kind of intricate network data that a real-world context discovery tool might generate. The great volume and complexity of nodes and their interconnections are a significant challenge for the connector's parsing and graph-building logic. The success of this test case validates that the connector's data handling is scalable and robust enough for a production environment.
The primary purpose of this example is to validate that the connector can correctly parse an extensive set of JSON input files. It then accurately reconstructs a complex Service Graph that maintains the integrity of all node relationships and roles. This test case ensures the script's robustness in generating an XML file that can be successfully processed by the VEREFOO framework. The PropertyDefinition section, as in other examples,

specifies the security policies to be enforced, including both ReachabilityProperty and IsolationProperty to test the verification of specific communication paths within the network.
The connector's ability to deduce the `functional_type` for each node, even without explicit declarations, is the feature that allows the automatic creation of a semantically correct network model. The bidirectional link validation ensured that connections were accurately represented, preventing logical errors that could lead to an incorrect verification result from VEREFOO. The resulting XML graph was a complete and coherent representation of the network topology.
The connector generates the following XML structure for this case:

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBCLIENT" name="10.17.2.108">
        <neighbour name="192.168.0.202"/>
      </node>
      <node functional_type="WEBCLIENT" name="192.168.0.202">
        <neighbour name="10.17.2.104"/>
        <neighbour name="10.17.2.106"/>
        <neighbour name="10.17.2.107"/>
        <neighbour name="10.17.2.108"/>
        <neighbour name="10.17.2.109"/>
        <neighbour name="20.0.0.1"/>
        <neighbour name="20.0.0.3"/>
        <neighbour name="20.0.0.4"/>
      </node>
      ...
      <node functional_type="WEBCLIENT" name="20.0.0.3">
        <neighbour name="192.168.0.200"/>
        <neighbour name="192.168.0.201"/>
        <neighbour name="192.168.0.202"/>
        <neighbour name="20.0.0.4"/>
      </node>
      <node name="20.0.0.4">
        <neighbour name="192.168.0.200"/>
        <neighbour name="192.168.0.201"/>
        <neighbour name="192.168.0.202"/>
        <neighbour name="20.0.0.3"/>
      </node>
    </graph>
  </graphs>
</NFV>
```

The security property definition file is shown below:

```
<NFV>
  <PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty"
    src="192.168.0.202" dst="10.17.2.104"/>
    ...
    <Property graph="0" name="ReachabilityProperty"
    src="10.17.0.234" dst="192.168.0.200"/>
    <Property graph="0" name="ReachabilityProperty"
    src="10.17.1.45" dst="192.168.0.201"/>
  </PropertyDefinition>
</NFV>
```

After the automated merge, the complete XML file submitted to VEREFOO, which now includes the property definition, is the following:

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBCLIENT" name="10.17.2.108">
        <neighbour name="192.168.0.202"/>
      </node>
      ...
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty" src="192.168.0.202"
        dst="10.17.2.104"/>
    ...
    <Property graph="0" name="IsolationProperty" src="192.168.0.202"
        dst="192.168.0.201"/>
    <Property graph="0" name="ReachabilityProperty" src="10.17.0.234"
        dst="192.168.0.200"/>
  </PropertyDefinition>
</NFV>
```

The file differentiation will not be shown in the next examples to avoid unnecessary redundancy.

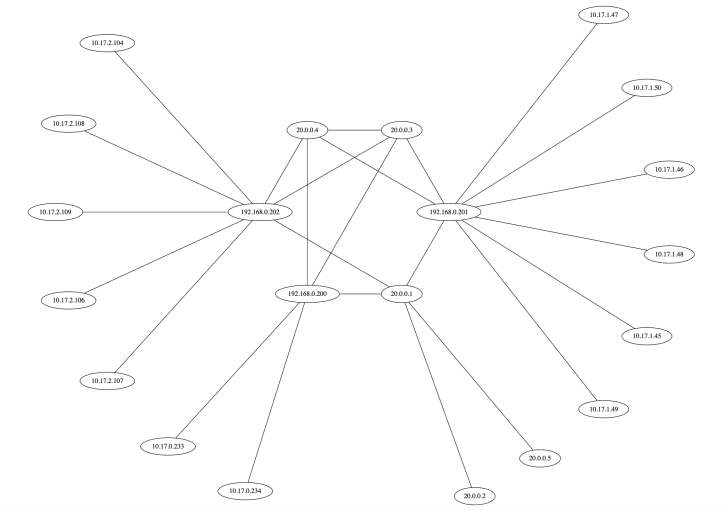

Figure 5.1. Network topology of Example 0

The VEREFOO API response of this simple topology is the following:

```
<Resources>
 <links>
  <links>
   <rel>self</rel>
   <href>http://localhost:8085/verefoo/adp/simulations/1</href>
   <hreflang/>
   <media/>
   <title>get the resource</title>
   <type/>
   <deprecation/>
  </links>
    ...
</Resources>
```

The final output file, which can be directly consumed by an OpenC2 actuator, in this simple case will be:

```
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("192.168.0.202"),
    dst_addr=oc2.IPv4Net("10.17.2.104")),
    arg, actuator=pf)
...
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("10.17.0.234"),
    dst_addr=oc2.IPv4Net("192.168.0.200")),
```

```
    arg, actuator=pf)
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("10.17.1.45"),
    dst_addr=oc2.IPv4Net("192.168.0.201")),
    arg, actuator=pf)
```

The following figure illustrates the terminal output for the execution of Example 0:

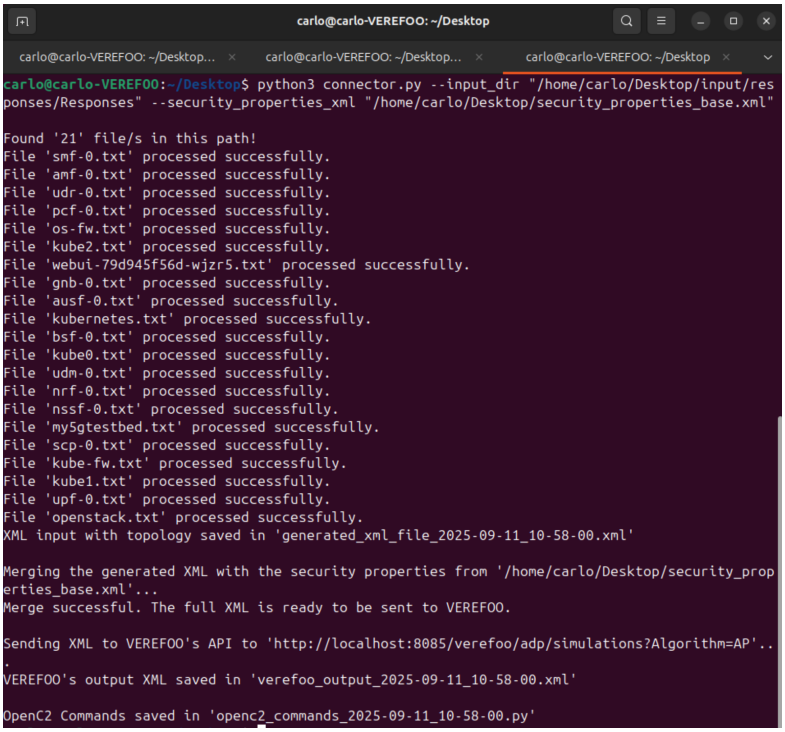

Figure 5.2. Terminal execution of Example 0

5.2.2 Example 1: Reachability and Isolation

To illustrate the validation process, it is used a simplified network topology. The topology, represented in the provided XML file, consists of a central firewall (20.0.0.1) connecting a web server (10.0.0.10) and three web clients (10.0.0.20, 10.0.0.30, and 10.0.0.40).
This test case was specifically designed to validate the connector's handling of core security properties, by defining a large number of ReachabilityProperty and IsolationProperty considering the small number of nodes. The simple topology allowed for a clear and unambiguous interpretation of the results.
The key element of this validation test case is the definition of security properties in the <PropertyDefinition> section, VEREFOO's formal verification results are determined by whether an explicit rule is needed to satisfy the property.

The connector generates the following XML file, which is then submitted to the VEREFOO API:

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBSERVER" name="10.0.0.10">
        <neighbour name="20.0.0.1"/>
      </node>
      ...
      <node name="20.0.0.1">
        <neighbour name="10.0.0.10"/>
        <neighbour name="10.0.0.20"/>
        <neighbour name="10.0.0.30"/>
        <neighbour name="10.0.0.40"/>
      </node>
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty" src="10.0.0.30"
        dst="10.0.0.40"/>
    ...
    <Property graph="0" name="IsolationProperty" src="20.0.0.1"
        dst="10.0.0.30"/>
    <Property graph="0" name="IsolationProperty" src="10.0.0.40"
        dst="20.0.0.1"/>
  </PropertyDefinition>
</NFV>
```

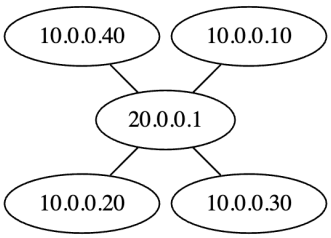

Figure 5.3. Network topology of Example 1

The VEREFOO API responds with a complete XML document, which the connector retrieves and parses.

```
<Resources>
 <links>
  <links>
   <rel>self</rel>
   <href>http://localhost:8085/verefoo/adp/simulations/1</href>
   <hreflang/>
   <media/>
   <title>get the resource</title>
   <type/>
   <deprecation/>
  ....
    </links>
 </content>
</Resources>
```

The final output file, which can be directly consumed by an OpenC2 actuator, is:

```
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("10.0.0.30"),
    dst_addr=oc2.IPv4Net("10.0.0.40")),
    arg, actuator=pf)
...
cmd = oc2.Command(oc2.Actions.deny, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("20.0.0.1"),
    dst_addr=oc2.IPv4Net("10.0.0.30")),
    arg, actuator=pf)
cmd = oc2.Command(oc2.Actions.deny, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("10.0.0.40"),
    dst_addr=oc2.IPv4Net("20.0.0.1")),
    arg, actuator=pf)
```

5.2.3 Example 2: Complex Topology

Another example of network topology consists of two distinct subnets: a DMZ (De-Militarized Zone) and an internal network. The DMZ is managed by a central firewall at 20.0.0.1 and includes a web server (172.16.0.10) and several web clients (172.16.0.20, 172.16.0.21, 172.16.0.22, and 172.16.0.23).
The internal network is managed by 10.0.0.254 and contains an internal web server (10.0.0.10), a web client (10.0.0.20), and additional web clients in different subnets (10.0.1.30, 10.0.1.31, 10.0.1.32, 10.0.2.33, and 10.0.2.34). This test case was designed to simulate a more realistic enterprise network architecture. The inclusion of a DMZ, multiple subnets, and inter-subnet routing points validates the connector's ability to accurately map more complex routing logic. These two main subnets are interconnected via 192.168.1.254

and 10.0.0.254. The key element of this validation test case is the definition of security properties in the <PropertyDefinition> section. The formal verification results will be determined by whether an explicit rule is needed to satisfy each property. The connector generates the following XML file, which is then submitted to the VEREFOO API: the properties defined for this topology were designed to test inter-subnet communication, for example, a ReachabilityProperty from a DMZ client to an internal web server.

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBCLIENT" name="172.16.0.21">
        <neighbour name="20.0.0.1"/>
      </node>
      <node functional_type="WEBCLIENT" name="10.0.1.32">
        <neighbour name="10.0.0.254"/>
      </node>
      ...
      <node name="10.0.0.254">
        <neighbour name="10.0.0.10"/>
        <neighbour name="10.0.0.20"/>
        <neighbour name="10.0.1.30"/>
        <neighbour name="10.0.1.31"/>
        <neighbour name="10.0.1.32"/>
        <neighbour name="10.0.2.33"/>
        <neighbour name="10.0.2.34"/>
        <neighbour name="192.168.1.254"/>
      </node>
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty" src="20.0.0.1"
        dst="172.16.0.10"/>
    ...
    <Property graph="0" name="IsolationProperty" src="10.0.1.31"
        dst="10.0.0.254"/>
    <Property graph="0" name="IsolationProperty" src="172.16.0.21"
        dst="172.16.0.10"/>
  </PropertyDefinition>
</NFV>
```

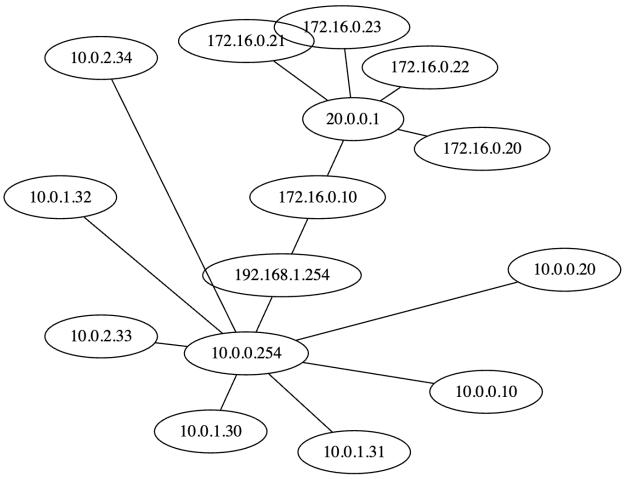

Figure 5.4. Network topology of Example 2

The VEREFOO API response of this more complex topology is the following:

```
<Resources>
 <links>
  <links>
   <rel>self</rel>
   <href>http://localhost:8085/verefoo/adp/simulations/1</href>
   <hreflang/>
   <media/>
   <title>get the resource</title>
   <type/>
   <deprecation/>
  </links>
    ....
</Resources>
```

The final output file, which can be directly consumed by an OpenC2 actuator, in this case will be:

```
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("20.0.0.1"),
    dst_addr=oc2.IPv4Net("172.16.0.10")),
    arg, actuator=pf)
...
cmd = oc2.Command(oc2.Actions.deny, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("10.0.1.31"),
    dst_addr=oc2.IPv4Net("10.0.0.254")),
```

```
    arg, actuator=pf)
cmd = oc2.Command(oc2.Actions.deny, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("172.16.0.21"),
    dst_addr=oc2.IPv4Net("172.16.0.10")),
    arg, actuator=pf)
```

One example of what is shown in the terminal is:

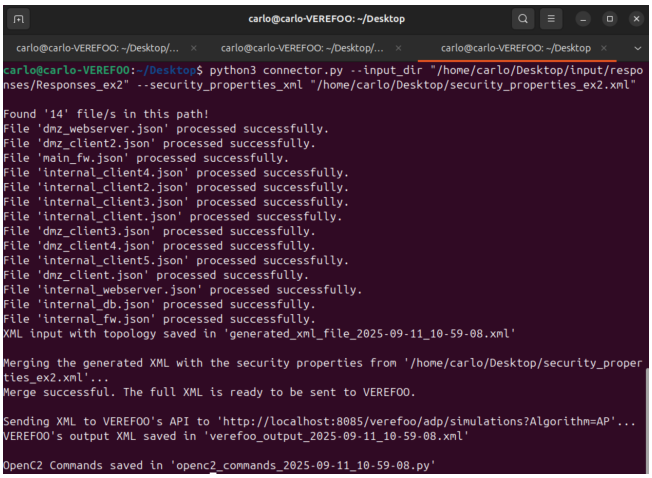

Figure 5.5. Terminal execution of Example 2

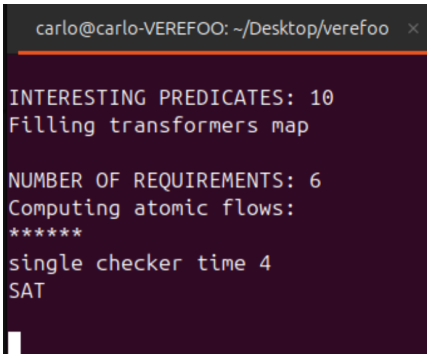

Figure 5.6. VEREFOO terminal of Example 2

5.2.4 Example 3: Connector Resilience

This example demonstrates a test case assessing the robustness and error-handling capabilities of the connector script. Instead of providing a complete and valid network topology, this test uses a set of input JSON files that are intentionally incomplete or malformed. This test case was designed to simulate the unpredictable nature of real-world data sources. By purposely introducing missing IP addresses and malformed links, we could verify that the connector's parsing logic is solid. The ability to handle these data anomalies is a measure of the tool's readiness for a production environment.
The simulated network consists of a basic topology with a firewall, a web server, and a web client, and the JSON input files contain deliberate errors to test the script's resilience: despite the errors in the input, the connector successfully assigned fallback IP addresses to the nodes that were missing them.

- The JSON file for the web server is missing a crucial field, its IP address, that will be assigned by the script.
- The JSON for the web client contains a malformed IP value.
- The firewall's JSON file has a missing or empty links array, preventing the correct mapping of connections to its neighbours.

The connector produces the following XML file, submitted to the VEREFOO API for verification:

```
<?xml version="1.0" ?>
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <node functional_type="WEBCLIENT" name="20.0.0.1">
        <neighbour name="10.0.0.2"/>
      </node>
      <node functional_type="WEBSERVER" name="20.0.0.2">
        <neighbour name="10.0.0.2"/>
      </node>
      <node name="10.0.0.2">
        <neighbour name="20.0.0.1"/>
        <neighbour name="20.0.0.2"/>
      </node>
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
    <Property graph="0" name="ReachabilityProperty" src="20.0.0.1"
        dst="20.0.0.2"/>
    <Property graph="0" name="IsolationProperty" src="10.0.0.2"
        dst="20.0.0.1"/>
  </PropertyDefinition>
</NFV>
```

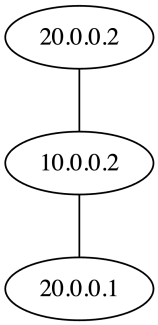

Figure 5.7. Network topology of Example 3

The VEREFOO API response of this basic topology is the following:

```
<Resources>
 <links>
  <links>
   <rel>self</rel>
   <href>http://localhost:8085/verefoo/adp/simulations/1</href>
   <hreflang/>
   <media/>
   <title>get the resource</title>
   <type/>
   <deprecation/>
      ...
  </links>
</Resources>
```

The final output file, which can be directly consumed by an OpenC2 actuator, in this simple case is:

```
cmd = oc2.Command(oc2.Actions.allow, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("20.0.0.1"),
    dst_addr=oc2.IPv4Net("20.0.0.2")),
    arg, actuator=pf)
cmd = oc2.Command(oc2.Actions.deny, oc2.IPv4Connection(
    src_addr=oc2.IPv4Net("10.0.0.2"),
    dst_addr=oc2.IPv4Net("20.0.0.1")),
    arg, actuator=pf)
```

This test case demonstrates that the connector can handle missing or malformed data while continuing to process valid input.

5.3 Analysis and Limitations

The validation results confirm that the connector successfully achieves its primary objective: providing an automated bridge between OpenC2 CTXD and VEREFOO. The tool demonstrates robust functionality in several key areas:

- Data Fidelity: the connector accurately translates the complex, nested JSON structure into a clean and coherent VEREFOO XML graph, ensuring that no network elements or links are omitted.
- Role Deduction: the heuristic-based `deduce_service_role` function proves to be effective in heuristic assigning functional types, which is critical for VEREFOO's verification logic.
- Graph Coherence: the two-pass approach and the bidirectional link check ensure that the generated XML represents a complete and valid network topology, preventing errors in VEREFOO's analysis.

- Actionable Output: the final step of generating OpenC2 commands from VEREFOO's response directly links formal verification to practical network security actions, closing the automation loop.

Chapter 6

Conclusions and Future Work

This chapter summarizes the key achievements of this thesis project, draws final conclusions, and outlines directions for future research. It serves as a synthesis of the entire project, connecting the theoretical background and problem statement with the practical implementation and validation. The first section provides a high-level overview of the successful implementation and validation of the connector, reflecting on how the initial objectives were met. The second section, dedicated to future work, discusses how the project's foundational work can be expanded to address more complex and dynamic cybersecurity challenges.

6.1 Conclusions

This thesis successfully implemented and validated a connector that bridges OpenC2 Context Discovery (CTXD) with the VEREFOO formal verification framework. The primary objective was to automate the security management loop by transforming fragmented network context data into actionable security policies [5]. As demonstrated in the Validation chapter, the connector proved its effectiveness in three key areas:

- Robust Data Translation: the connector successfully parses complex, nested OpenC2 JSON data, handling incomplete or malformed inputs, and generates a syntactically and semantically valid VEREFOO XML file. The multi-pass approach and automated role deduction proved to be effective in creating a complete and verifiable representation of the network, even with a minimal starting dataset. This robust data handling is achieved through several key technical decisions: the `get_nested_value` function was specifically implemented to correctly traverse nested JSON objects, preventing errors from missing or improperly structured data; the `deduce_service_role` function autonomously categorizes network entities (e.g., webserver, forwarder, nat), ensuring that the VEREFOO XML output accurately reflects the functional role of each node without requiring manual input. The multi-pass approach and automated role deduction proved to be effective in creating a complete and verifiable representation of the network, even with a minimal starting dataset. This process includes a sophisticated fallback mechanism for assigning unique IP

addresses to nodes lacking this information, using an arbitrary range (20.0.0.1 to 20.255.255.255) to guarantee graph completeness and prevent IP collisions, which is a critical requirement for formal verification tools like VEREFOO. It also ensures the bidirectionality for the neighbours, as requested by VEREFOO.

- Automated Data Consolidation: the connector automatically merges the generated XML network topology with a user-defined file containing security properties. This process eliminates the need for manual file editing, ensuring a single, seamless execution of the script from start to finish. This approach prevents premature workflow termination and streamlines the entire security analysis pipeline, making it more efficient and less prone to human error.

- Integration with Formal Verification: the connector's ability to communicate with the VEREFOO API ensures that the formal verification process is an automated step. This communication is powered by the Python *requests* library, which allows the script to send a POST request containing the generated XML. After the initial POST, the connector checks the response for a results URL, and if one is provided, it then performs a GET request to retrieve the final verification results when they are ready. The successful processing of different topologies, from simple client-server setups to complex DMZ configurations, confirms that the generated XML is a reliable input for VEREFOO's analysis and that the connector's API logic is resilient and effective.

- Closing the Security Automation Loop: generating the OpenC2 commands based on VEREFOO's output and translating isSat="true" results into corrective allow or deny commands, the connector transforms security analysis into an executable response. This effectively closes the loop between network discovery, formal policy verification, and automated security enforcement.

This thesis project provides a proactive and verifiable approach to network security that replaces manual, reactive security interventions with an automated, logic-based system that can identify, analyze, and remediate policy violations at scale, strengthening network security and reducing the potential for human error [13]. This work serves as a proof-of-concept for the practical application of formal methods in cybersecurity, demonstrating that complex verification can be integrated into an automated, operational workflow to enhance network defense. Specifically, this work directly addresses the need to integrate formal methods into automated security workflows, a challenge highlighted in the research on the state of the art of network security automation [2].

6.2 Future Work

The implemented connector provides a solid foundation for further research and development in automated network security orchestration. While effective, the current implementation can be expanded in several key areas to enhance its capabilities, scalability, and integration with real-world systems.

- Handling State and Dynamic Context: the connector processes a static snapshot of network context. A significant improvement would be to adapt the tool to a dynamic environment. This could involve implementing a continuous polling mechanism to react to real-time changes in the network, such as a new host coming online or a link failing. This would require incorporating a persistent data store to maintain the network's state over time and handle incremental updates.
- Full OpenC2 Actuator Integration: another possible next step would be the development of a fully functional OpenC2 Actuator module that can directly execute these commands on a live network device (e.g., an iptables or a cloud-based firewall API). This module could be a dedicated service that listens for the OpenC2 commands generated by the connector and translates them into specific API calls for real-world network devices or cloud-based firewalls. This would eliminate the manual step of executing the generated commands, achieving a truly end-to-end automated pipeline from discovery to enforcement.

Bibliography

[1] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. Automated optimal firewall orchestration and configuration in virtualized networks. 2020.

[2] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. Automation for network security configuration: state of the art and research trends. 2024.

[3] Vasileios Mavroeidis and Joe Brule. A nonproprietary language for the command and control of cyber defenses – openc2. 2020.

[4] Gianmarco Bachiorrini, Daniele Bringhenti, and Fulvio Valenza. Adaptive, agile and automated cybersecurity management. 2025.

[5] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. A looping process for cyberattack mitigation. 2024.

[6] Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. Security automation in next-generation networks and cloud environments. 2024.

[7] OASIS OpenC2 Technical Committee. Open command and control (openc2) architecture specification. Technical report, OASIS, 2022.

[8] OASIS OpenC2 Technical Committee. Open command and control (openc2) language specification. Technical report, OASIS, 2024.

[9] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. A novel abstraction for security configuration in virtual networks. *Computer Networks*, 2023.

[10] OASIS OpenC2 Technical Committee. Specification for json abstract data notation (jadn). Technical report, OASIS, 2021.

[11] Silvio Tanzarella. Developing the context discovery actuator profile for openc2 language. Master's degree thesis, Politecnico di Torino, 2024.

[12] OASIS OpenC2 Technical Committee. Specification for transfer of openc2 messages via https. Technical report, OASIS, 2021.

[13] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. Atomizing firewall policies for anomaly analysis and resolution. *IEEE Transactions on Dependable and Secure Computing*, 2025.

[14] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. Towards security automation in virtual networks. 2023.

[15] Daniele Bringhenti, Lucia Seno, and Fulvio Valenza. An optimized approach for assisted firewall anomaly resolution. *IEEE Access*, 2023.

[16] Gianmarco Bachiorrini, Daniele Bringhenti, and Fulvio Valenza. Toward the optimization of automated vpn configuration. 2025.

[17] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. Automated firewall configuration in virtual networks. *IEEE Transactions on Dependable and Secure Computing*, 2023.

[18] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Serena Spinoso, Fulvio Valenza, and Jalolliddin Yusupov. Improving the formal verification of reachability policies in virtualized networks. *IEEE Transactions on Network and Service Management*, 2021.

[19] Riccardo Sisto, Guido Marchetto, Fulvio Valenza, Daniele Bringhenti, and Jalolliddin Yusupov. Towards a fully automated and optimized network security functions orchestration. 2019.

[20] Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. A demonstration of verefoo: an automated framework for virtual firewall configuration. 2023.

[21] Daniele Bringhenti, Francesco Pizzato, Riccardo Sisto, and Fulvio Valenza. Autonomous attack mitigation through firewall reconfiguration. *International Journal of Network Management*, 2024.

[22] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. Introducing programmability and automation in the synthesis of virtual firewall rules. 2020.

[23] Daniele Bringhenti, Jalolliddin Yusupov, Alejandro Molina Zarca, Fulvio Valenza, Riccardo Sisto, Jorge Bernal Bernabé, and Antonio F. Skarmeta. Automated, verifiable and optimized policy-based security enforcement for sdn-aware iot networks. *Computer Networks*, 2022.