POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Remediation procedures and automated cybersecurity incident response

Supervisors

Prof. Cataldo Basile

Dott. Francesco Settanni

Candidate

Dario Simone Leone

Academic Year 2024/2025

Torino

Abstract

Digital transformation in recent years has fostered the adoption of interconnected technologies, such as scalable cloud services, the pervasive Internet of Things, and artificial intelligence, altering the approach organizations take to their routine operations. While these advancements bring benefits, they also expand the attack surface, exposing organizations to more frequent and sophisticated cybersecurity threats. Attackers leverage emerging technologies to orchestrate targeted campaigns, highlighting the need for automated and standardized Incident Response processes. Despite efforts to improve automation, the diversity of attack types and environments, spanning traditional information technology, cloud platforms, and industrial control systems, makes one-size-fits-all solutions impractical. There is thus a growing need to abstract response procedures from specific technologies and encode them for interoperability without constant manual adaptation. Standardized formats for representing incident response actions facilitate automation, integration, and transformation of heterogeneous procedures into homogeneous playbooks. Such an approach would enable teams to focus on strategic decision-making while routine actions are handled automatically, reducing response times. This thesis presents a framework for procedural and automated remediation based on security playbooks, designed within the Security Orchestration, Automation, and Response (SOAR) paradigm. This approach also enables the subsequent sharing of remediation strategies, similarly to Cyber Threat Intelligence (CTI). The proposed tool maps alerts to corrective actions defined in structured playbooks and translates them into executable instances, preserving complex logical structures such as conditions, loops, and parallel actions. A novelty of this work is the design and implementation of a security automation framework tailored for modern cloud-native environments, such as Kubernetes clusters, in addition to traditional on-premises infrastructure.

Table of Contents

List of Figures

Chapter 1

Introduction

In an increasingly interconnected digital world, the ability to respond effectively to cybersecurity incidents is a crucial requirement for ensuring the resilience and continuity of modern infrastructures. From enterprise environments to governmental institutions, the frequency and severity of cyberattacks have escalated, placing enormous pressure on security teams and traditional defense mechanisms. Advanced persistent threats, zero-day vulnerabilities, and large-scale ransomware campaigns illustrate the complexity and impact of modern cyber incidents, often overwhelming manual remediation processes. Risks stem not only from malicious attacks but also from accidents, misconfigurations, and the increasing complexity of interdependent infrastructures. In this evolving threat landscape, the discipline of incident response has gained strategic importance. Timely detection, containment, and remediation are not only essential for limiting damage but also for restoring trust and operational capacity. However, conventional incident response methods are increasingly challenged by the scale, speed, and sophistication of attacks. Manual procedures are often time-consuming and prone to errors, leading to delayed reactions and increased exposure. To address these challenges, automation has emerged as a key enabler in designing scalable and efficient incident response frameworks. Automated incident response systems aim to minimize human intervention in repetitive and well-defined tasks, enabling security teams to concentrate on complex decision-making. By integrating detection mechanisms, predefined remediation workflows, and contextual analysis, such systems can significantly enhance the effectiveness and speed of the response process.

1.1 Motivation and Research Drivers

In today's digital landscape, organizations are facing a constant increase in the frequency, complexity, and impact of cyber threats. These threats range from sophisticated attack campaigns, including Advanced Persistent Threat (APT), to targeted ransomware and multi-stage intrusions. Such attacks demand rapid and structured response mechanisms.

Traditional Incident Response (IR) processes heavily rely on human expertise and predefined procedures. However, this model has become insufficient in dynamic environments

where speed and scalability are critical. Moreover, incident response is often perceived as an additional burden for security personnel, such as Security Operations Centre (SOC) analysts, who are already responsible for a wide range of operational tasks.

Security Orchestration, Automation, and Response (SOAR) platforms have emerged as a strategic approach to tackle these challenges by automating IR tasks. Nevertheless, the effectiveness of these systems depends on the availability of actionable information, the ability to accurately map threats to corresponding response actions, and the integration across heterogeneous Information Technology (IT) environments. Several studies emphasize the importance of refining the selection, adaptation, and execution of remediation procedures in response to complex incidents [1].

The motivation behind this thesis stems from the growing need to support incident response teams with a reliable and automated tool that reduces manual workload and response latency. The proposed solution aims to automate key response activities, such as isolating affected systems, blocking malicious IP addresses, or activating containment workflows, depending on the nature of the incident.

Recent research also emphasizes the importance of contextual awareness [2], threat intelligence integration, and orchestration flexibility to enhance cyber resilience.

The main challenges addressed in this work include:

- **Consistent threat-to-remediation mapping**, ensuring coherent alignment between detected threats and defensive actions.
- **Automation of remediation procedures** with a focus on scalability and extensibility, enabling integration with heterogeneous environments;
- **Design for usability and orchestration**, creating a tool aligned with common practical strategies in incident response while supporting customization;

1.2 Objectives

This thesis aims to design a tool that supports end-to-end incident response, from alert reception to the execution of remediation actions. The tool automates the transformation of diverse procedures into standardized, executable playbooks, improving efficiency, consistency, and operational readiness.

Abstraction and automation enable the system to scale across various environments, including cloud infrastructures, while compatibility with standards such as Collaborative Automated Course of Action Operations (CACAO) facilitates seamless integration and interoperability. In addition to executing response actions, the tool provides reporting capabilities to document and analyze incident handling, supporting accountability and continuous improvement.

By addressing these challenges, this work contributes to the advancement of automated incident response solutions, aiming to enhance both the efficiency and reliability of security operations while minimizing operational constraints.

1.3 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2 presents the current scenario of incident response and the relevant standards, providing the foundation for the research described in subsequent chapters.

Chapter 3 reviews related works, beginning with the original tool from which this thesis was developed and then discussing other tools compliant with the standards, highlighting solutions that inspired new approaches in the development of the framework.

Chapter 4 positions the proposed framework within the broader context of incident response and identifies the current challenges it aims to address.

Chapter 5 details the design of the tool, including the libraries used, the class structure, and the overall architectural design.

Chapter 6 focuses on the implementation of the proposed solutions, providing a detailed account of how the design choices were realized.

Chapter 7 discusses the validation of the framework, describing the real-world scenarios used for testing and the actions executed during these tests.

Chapter 8 presents conclusions and possible future work related to this project.

Appendix A offers a user guide for running and interacting with the tool.

Appendix B provides a technical guide for developers who wish to extend or better understand its inner workings.

Chapter 2

Background

This chapter introduces the evolving landscape of incident response, from the foundational principles outlined by European Union Agency for Cybersecurity (ENISA) to the modern SOCs. It explores how organizations address incidents across traditional and modern infrastructures, including Industrial Control Systems (ICSs) and Software-Defined Networking (SDN) environments, where automation becomes increasingly essential. In this context, the use of structured response mechanisms, such as playbooks, is discussed, with a particular focus on standardized formats, which will be examined in detail in the following sections.

2.1 Introduction to Incident Response: From Critical Infrastructure to SOC Operations

Modern society relies deeply on communication networks and information systems. As such, ensuring the security and continuous operation of these systems has become a priority for maintaining economic stability, delivering public services, and promoting societal well-being.

In response to these challenges, the European Union established the ENISA. The mission of ENISA is to enhance network and information security in the Member States, support the development of Computer Emergency Response Teams (CERTs), also known as Computer Security Incident Response Teams (CSIRTs), and cultivate a security-aware ecosystem that benefits citizens, enterprises, and public institutions.

2.1.1 Role of CERTs/CSIRTs in Incident Response

Central to the EU's incident response framework are both national and organizational teams, commonly referred to as CERTs or CSIRTs. These terms are generally used interchangeably [3] to indicate specialized units dedicated to managing cybersecurity incidents. National teams often serve as reference points for governments and critical sectors, while organizational teams are established within companies or institutions to

protect their own infrastructures. Together, they represent the first line of defence for critical information systems.

Their activities encompass the entire incident lifecycle, from detection and triage to response coordination, provision of mitigation guidance, post-incident analysis, and promotion of security awareness within their constituency. The notion of *constituency* is fundamental in incident management, as it defines the scope of responsibility of each team, such as a set of IP ranges, domains, autonomous systems, or the members of a specific organization. Once this constituency is established, CERTs/CSIRTs are expected to engage actively with it through security announcements, early threat alerts, and participation in trusted communities, thereby fostering awareness, timely reporting, and effective collaboration during crises.

ENISA guidance highlights the importance of clearly defined and coordinated constituencies [3], even when overlaps occur. Each incident response organization should clearly define its mandate, specify the types of incidents it handles, and outline the expectations it has of its stakeholders, thereby reducing ambiguity and ensuring efficient cooperation.

2.2 Incident Response Process According to ENISA

The ENISA outlines a structured approach to incident response, which includes several key phases (Figure 2.1).

2.2.1 Incident Report

The incident response process begins when the CERT receives a report about a potential security incident. This report can be submitted through various communication channels, including e-mail, phone, fax, postal mail, walk-in reports, or web forms. In practice, e-mail is the most common and preferred method, particularly because it can be easily integrated with automated incident handling systems.

2.2.2 Registration

Once received, the incident is formally registered in the incident handling system. Each incident is assigned a unique identifier, typically following a format such as `[CERT-NAME# report_number]`, to facilitate tracking and future reference. If the new report is related to a previously registered incident, it may be linked or merged accordingly.

2.2.3 Triage

The triage phase involves determining the priority and urgency of the incident. It consists of three sub-phases:

- **Verification:** Confirming that the reported issue is indeed a legitimate incident.
- **Initial Classification:** Identifying the type and severity of the incident.

- **Assignment:** Allocating the incident to an appropriate team or individual for further handling.

2.2.4 Incident Resolution

Following triage, the resolution phase begins. This phase is typically the most time-consuming and includes several iterative steps:

- **Data analysis.**
- **Researching possible solutions.**
- **Proposing actions.**
- **Executing actions.**
- **Eradication of the threat and recovery of affected systems.**

2.2.5 Incident Closure

Once resolution is achieved, the incident is formally closed. Best practices recommend performing a final verification, documenting the incident thoroughly, and, optionally, obtaining feedback from the stakeholders involved.

2.2.6 Post-analysis

After closure, a post-analysis phase is conducted to reflect on the incident and the response process. This phase is typically done after some time has passed, allowing the team to revisit the incident with a fresh perspective. The aim is to identify lessons learned, improve procedures, and enhance overall readiness for future incidents.

2.3 From CERTs to Security Operations Centres

As the volume and complexity of cyber threats continue to increase, organizations have moved beyond national CERTs and CSIRTs structures by establishing internal SOCs. While traditional CERTs emphasizes coordination and external communication, SOCs acts as a centralized, real-time operational unit within organizations.

Modern SOCs build upon the foundational principles introduced in CERTs frameworks, such as constituency awareness, structured workflows, triage, and escalation, and enhance them through the integration of automated tools, real-time telemetry, and tailored response strategies.

Although incident detection typically falls outside the scope of incident management, it is essential to note that the contemporary incident response process in SOC environments begins immediately after detection. The process can be broadly divided into four key phases: detection, triage, analysis, and response.

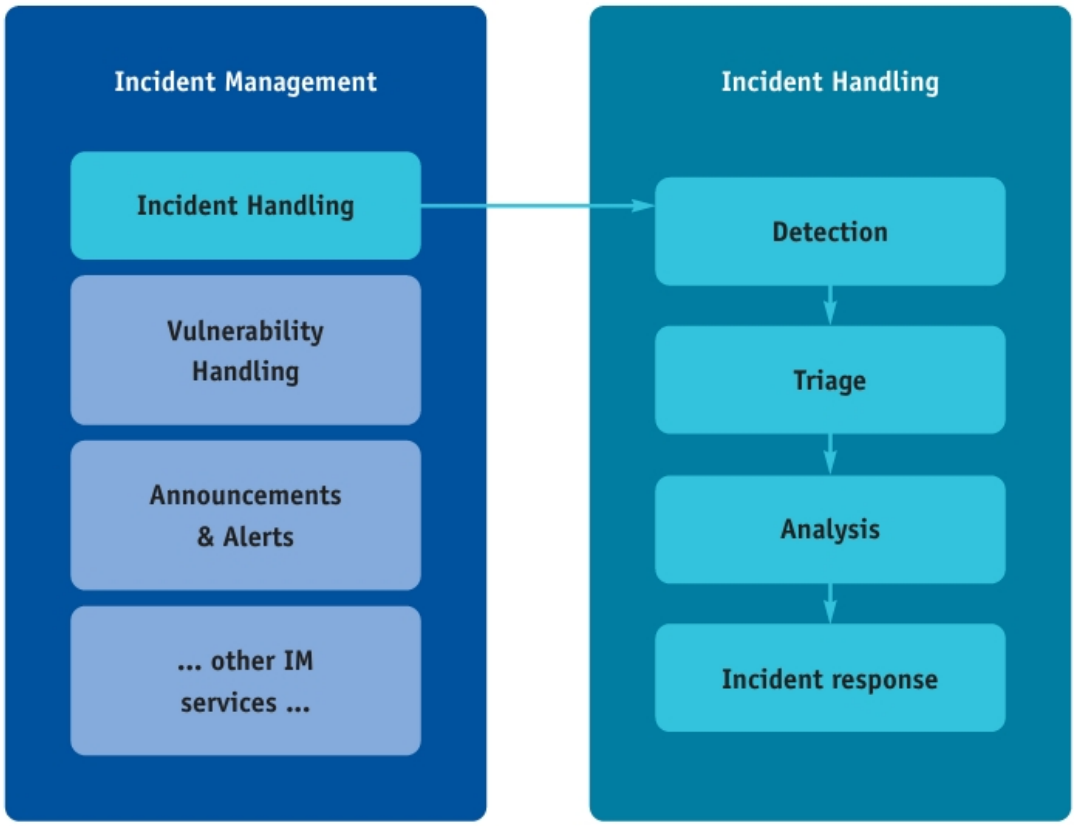

Figure 2.1: Incident management and incident handling.[3]

During the triage phase, reported incidents are initially verified to eliminate false positives, such as spam or irrelevant alerts. Verified incidents are then classified according to internal taxonomies and prioritized based on severity and impact. Finally, they are assigned to specific teams or analysts for further handling.

SOCs serve as the nerve center for proactive cyber defence and incident response. SOC teams are responsible for continuous monitoring, triage, investigation, containment, and coordinated response to security incidents. Analysts ranging from junior Level 1 triage staff to senior threat hunters and incident responders work alongside escalation managers and threat intelligence operators. Modern SOCs centralize telemetry from Security Information and Event Management (SIEM), Intrusion Detection Systems (IDSs)/Intrusion Prevention Systems (IPSs), endpoint detection tools, threat intelligence feeds, and orchestration platforms, ensuring that incidents can be detected, contextualized, and managed efficiently.

While SOCs started as reactive monitoring units, they are now critical for maintaining organizational resilience by reducing response times, enforcing consistent playbooks, and enabling internal and external collaboration. Automation and standardized procedures help avoid fragmented workflows and ensure clarity in roles and responsibilities during an incident response cycle.

2.4 Evolution of SOCs: From SOC 1.0 to SOC 3.0

The SOC teams have evolved through three distinct generations, each driven by the need for greater efficiency, accuracy, and scalability in threat detection and response [4].

2.4.1 SOC 1.0: Manual Operations and Low Scalability

Early SOCs (SOC 1.0) were entirely manual. Alert triage, investigation, and response relied on human analysts interpreting static SIEM rules and correlating data from disparate sources. Detection was reactive based on known signatures, which led to many false positives, slow incident response, and overwhelmed analysts. Scalability was limited, response times were slow, and the risk of missed or delayed incidents was high.

2.4.2 SOC 2.0: Partial Automation with SOAR and XDR

In response to growing alert volume and complexity, SOCs 2.0 integrated SOAR and Extended Detection and Response (XDR). This enabled:

- Automatic alert enrichment via threat intelligence feeds;
- Improved event correlation across multiple data sources;
- Playbook-driven automation: for example, blocking IPs or disabling compromised credentials, while final decisions still require human approval.

It reduced manual workload and improved consistency.

2.4.3 SOC 3.0: AI-Powered Detection, Investigation and Response

SOC 3.0 marks a radical transformation: the integration of artificial intelligence (AI) into detection, triage, investigation and response. Key innovations include:

- **AI-based adaptive detection**: machine learning models continuously analyze and refine detection rules to identify emerging threats with lower false positive rates;
- **Automated investigation**: AI correlates events from multiple sources in real time, enabling junior analysts to handle cases that previously required senior experts;
- **Contextual response automation**: AI assesses the context of incidents and recommends—or even executes—mitigation steps, significantly reducing time-to-response while preserving decision quality;
- **Distributed data lakes and cost optimization**: rather than relying on a single SIEM, SOC 3.0 leverages distributed storage and on-demand querying to scale and reduce operational costs.

Table 2.1: SOC evolution

Feature	SOC 1.0	SOC 2.0	SOC 3.0
Alert triage	Manual	Enrichment + partial automation	AI-driven classification
Detection	Static SIEM rules	XDR-enabled correlation	Adaptive ML-driven detection
Investigation	Manual	Guided by enrichment but human-led	Automated deep-dive via AI
Response	Manual Standard Operating Procedures	Playbook execution with human oversight	Context aware AI suggestions and autonomous actions
Data storage	Centralized SIEM only	SIEM + siloed storage	Distributed data lakes with on-demand queries
Team structure	Tiered pyramid	Still tiered, less overload	Flattened, role based specialization

2.4.4 Recommender Systems in SOCs: A Tiered Approach

Recent research highlights the increasing relevance of recommender systems in enhancing decision-making within SOCs. In particular, the work titled *SoK: Applications and Challenges of Using Recommender Systems in Cybersecurity Incident Handling and Response* [5] presents a comprehensive systematic review (SoK) on the applications and challenges of recommender systems in the handling and response of cybersecurity incidents.

The authors introduce a structured four-tier model of SOC responsibilities, designed to reflect the internal hierarchy and functional granularity typical of modern security operations (Figure 2.2):

- **Tier 1 (Operational):** responsible for initial triage and execution of predefined response playbooks. This tier is characterized by time-sensitive, rule-driven tasks that benefit from automation and pattern-based decision support.
- **Tier 2 (Analytical):** conducts deeper investigations and correlates data with threat intelligence sources. This layer focuses on understanding incident scope and identifying Indicator of Compromises (IoCs) across systems.
- **Tier 3 (Strategic):** tasked with advanced threat hunting, managing complex or persistent incidents, and conducting proactive forensic analysis.
- **Tier 4 (Governance):** oversees SOC performance, ensures compliance with audit and escalation procedures, and manages long-term operational metrics.

For each tier, the paper outlines specific recommender system capabilities aligned with operational needs:

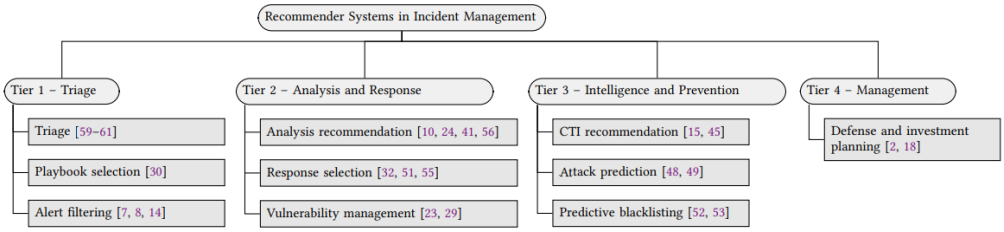

Figure 2.2: Taxonomy of recommender systems in cybersecurity [5].

- **Tier 1:** Suggests appropriate playbooks based on event features, time-of-day patterns, and past incident data using collaborative filtering or rule-based inference.
- **Tier 2:** Ranks candidate root causes and correlates alerts with historical attack patterns or threat intelligence feeds, employing probabilistic models or graph-based similarity.
- **Tier 3:** Generates hypotheses for threat hunting campaigns by identifying outlier behaviors or weak signal correlations.
- **Tier 4:** Provides dashboards for strategic decision support, summarizing incident trends, analyst performance, and system response times using aggregated historical data and visual analytics.

Moreover, the authors emphasize the technical challenges involved in implementing these systems, including:

- *Data sparsity and labeling issues*, especially in early-stage triage.
- *Evaluation complexity*, due to the absence of standardized performance metrics tailored to cyber incident handling.
- *Human-in-the-loop design*, where analyst feedback must be incorporated in real-time.

2.5 Industrial Control Systems versus Cloud-Native Environments

This section contrasts two critical operational domains: traditional ICSs and modern cloud-native/SDN environments (such as Kubernetes and multi-cluster orchestration). They will be analyzed in terms of distinct constraints, threat models, and appropriate remediation strategies.

2.5.1 Industrial Control Systems (ICS): Architecture, Security, and Challenges

ICSs are specialized architectures designed to monitor and control physical industrial processes such as energy production, water treatment, chemical manufacturing, and transportation. ICS is a broad term encompassing Supervisory Control and Data Acquisition (SCADA) systems, Distributed Control System (DCS), and Programmable Logic Controller (PLC), each tailored for different layers of industrial process automation [6].

An ICS typically includes a combination of electrical, mechanical, and digital components working together to achieve specific industrial objectives. Control logic can be automated or manual, and systems can operate in either open-loop or closed-loop configurations. Human Machine Interfaces (HMIs), sensors, actuators, and network communication protocols enable real-time monitoring, diagnostics, and process control.

Historically, ICS environments were air-gapped and operated on proprietary hardware and protocols, with little resemblance to conventional IT networks. However, in recent years, low-cost Ethernet and IP-based components have been widely adopted to improve interoperability and remote access. This convergence between IT and Operational Technology (OT) has increased the exposure of ICS to cyber threats, leading to a growing need for cybersecurity frameworks tailored explicitly to these environments [6].

Unlike traditional IT systems, ICSs have unique operational constraints:

- **Safety and availability take precedence over confidentiality**. System downtime may endanger human lives, damage equipment, or disrupt critical infrastructure.
- **Legacy components and proprietary protocols** are still widely used, often without active vendor support.
- **Patching is constrained** by real-time availability requirements and the potential risks introduced by system changes.

These constraints impose significant limitations on the application of conventional IT security solutions to ICSs. For example, antivirus software, frequent patching, and aggressive scanning may not be feasible due to the risk of service interruption. ICS security requires solutions that respect operational continuity, deterministic timing, and process safety.

Modern approaches to vulnerability management.

ICS/OT systems are governed by strict uptime requirements, making traditional patch management impractical. Alternative mitigation techniques [7], such as firewall reconfiguration or VPN segmentation, are often needed. The authors propose an automation pipeline that converts standardized Common Security Advisory Framework (CSAF) advisories into CACAO playbooks.

In their proof of concept, 79 CACAO-compliant playbooks were generated, covering 485 remediation steps. These playbooks enable repeatable and systematic vulnerability responses without requiring direct system modification—essential in ICS environments

where unplanned downtime is unacceptable. Such strategies are key to enabling SOCs to handle vulnerabilities in a structured and non-invasive manner, while ensuring process continuity and operational safety.

2.5.2 Cloud-Native/SDN Environments: Dynamic Infrastructure and Policy Automation

The growing adoption of SDN and cloud-native paradigms, such as container orchestration platforms (e.g., Kubernetes), has introduced a new layer of complexity for network security management. These environments are inherently dynamic: workloads are often ephemeral, distributed across namespaces, and interconnected through virtual networks.

In Kubernetes-based infrastructures, for example, security policies must regulate traffic between pods, services, and external endpoints. The fine-grained control required at this level is difficult to maintain manually, as policy definitions can become inconsistent or conflict-prone across namespaces. Manual authoring of network policies is not only error-prone but also fails to scale with the increasing velocity and complexity of deployment pipelines.

Recent automation platforms such as ARMO, and approaches leveraging extended Berkeley Packet Filter (eBPF), have emerged to address this challenge. These tools monitor real-time runtime behavior and automatically infer Kubernetes Network Policies based on observed communications and workload profiles. These platforms reduce the human burden and increase policy accuracy by reflecting actual behavior, rather than intended behavior.

In parallel, research in SDN environments—such as SDN Unified Policy Configuration (SUPC) and Zero Trust Software-Defined Networking (ZT-SDN)—has explored high-level policy abstraction and translation into enforceable flow rules. These systems aim to resolve policy conflicts, enforce compliance, and support secure service function chaining across multi-domain networks.

The key principles underpinning these frameworks are:

- **Intent-based policies**: Abstracting security goals at a high level (e.g., "App A must not talk to App B") and translating them into concrete flow rules or firewall entries.
- **Conflict detection and resolution**: Using formal verification or graph-based models to ensure that dynamically generated policies do not introduce security gaps or contradictions.
- **Integration with orchestration tools**: Enabling continuous security enforcement that evolves with infrastructure changes (e.g., pod rescheduling, auto-scaling).

Cilium vs. KubeArmor

A further dimension of cloud-native security is the ability not only to generate policies automatically but also to enforce remediation actions in real-time. Automation of responses ensures that security controls continually adapt to the evolving runtime context of

workloads, thereby reducing the time it takes to react to suspicious or malicious events.

Two representative approaches in this area are **Cilium** and **KubeArmor**, which complement each other by targeting different enforcement layers:

- **Cilium** operates primarily at the network layer, providing L3/L4/L7 enforcement through eBPF. It integrates seamlessly with Kubernetes as a Container Network Interface (CNI), enabling identity-aware filtering, fine-grained network policies, and service-mesh-level features such as load balancing and observability.
- **KubeArmor** instead focuses on runtime system-level protection, restricting file access, process execution, and kernel capabilities within pods. This element enables the blocking of unauthorized internal behaviors, even when network policies alone would be insufficient.

Together, these frameworks illustrate how automated responses in cloud-native environments can span multiple layers, from controlling network flows across services to constraining low-level process activity within containers. Such capabilities bring remediation closer to the runtime itself, making enforcement both fine-grained and adaptive.

2.6 Security Orchestration Automation and Response

SOAR represents a unified platform that enables SOCs teams to centralize security tool integration, automate routine tasks, and orchestrate coordinated incident response workflows. SOAR helps security operations deal with alert overload, tool sprawl, and staff constraints by streamlining investigation and response from a single console [8].

2.6.1 Core Capabilities

- **Security Orchestration:** SOAR platforms integrate diverse security tools (SIEM, Endpoint Detection and Response (EDR), threat intelligence feeds, sandboxing, firewalls, etc.) via APIs and prebuilt connectors, allowing for coordinated workflows across tools.
- **Security Automation:** The system automates repetitive tasks such as alert enrichment, ticket generation, response actions (e.g., isolating endpoints via Network Detection and Response (NDR), triggering antivirus scans), and playbook execution, significantly reducing manual workload.
- **Incident Response Management:** SOAR acts as a centralized IR console where analysts correlate alerts, filter out false positives, prioritize incidents, and invoke appropriate playbooks— all within one dashboard.

2.6.2 Benefits for Efficiency

IBM highlights several major advantages:

- **Accelerated response times**—by enriching alerts and executing dynamic playbooks, specific clients achieved up to an 85% reduction in incident response time.
- **Resource optimization**—SOAR relieves analysts from routine triage, enabling them to focus on complex investigation and threat hunting.
- **Consistency and centralized orchestration** — a unified hub ensures standardized workflows, clearer audit trails, and coordinated team actions.

2.6.3 Architectural Considerations

Modern SOAR platforms support flexible deployment—on-premises, cloud, or hybrid—to adapt to evolving security infrastructure. Open architectures and standards-based integrations maximize compatibility with existing tools, reduce vendor lock-in, and support centralized case management and collaboration across teams.

2.6.4 Auditability and Access Control

SOAR platforms maintain detailed audit logs to track every action and configuration change. This platform enhances forensic capability and supports compliance, as analysts can reconstruct workflows and identify misconfigurations or failures.

Attribute-Based Access Control (ABAC) enables granular, context-aware permission models—policies can restrict playbook execution or elevated privileges based on user attributes, time, or role, implementing least-privilege security without inhibiting workflow flexibility.

2.6.5 Case Study and Cross-Environment Comparison

A recent evaluation [9] integrating Splunk Enterprise SIEM with SOAR in an on-premises, sandboxed architecture demonstrated clear improvements in security posture, including more effective alert triage, faster automation-driven responses, and overall reduction in analyst burden.

2.6.6 Applicability to ICS/OT vs. SDN/Cloud Environments

In ICS/OT environments, where availability, safety, and process continuity are paramount, SOAR playbooks must be carefully designed to prevent unintended disruptions. Automation is valuable but must be constrained and paired with heightened human oversight. Alert enrichment from industrial sensors and controlled orchestration of incident response help SOCs manage OT incidents more reliably.

Conversely, in SDN or cloud-native environments, programmability and elasticity provide greater flexibility. Here, SOAR can automate responses aggressively—e.g., auto-scaling environments, isolating compromised workloads, or deploying rapid policy changes across infrastructure—and integrate with continuous deployment pipelines.

The Splunk SIEM/SOAR [9] study underscores this contrast: while hybrid SIEM/SOAR deployments in traditional IT allowed improved incident response workflows, adapting these workflows for ICSs requires specialized data sources and conservative response logic.

To address this, organizations increasingly rely on structured and automated procedures known as **playbooks**, which define predefined sequences of actions to enforce or remediate policies consistently across environments. Playbooks play a crucial role in streamlining operations and reducing the potential for human error, especially in dynamic infrastructures like Kubernetes. This thesis will examine cloud/SDN useful playbooks in detail, highlighting their structure, operational semantics, and their integration within automated policy enforcement workflows.

2.7 The Role of Cyber Threat Intelligence and Playbooks in Incident Response

Cyber Threat Intelligence (CTI) is a foundational element of modern security operations, informing decisions from detection through remediation. Schlette et al. [10] examine how CTI data models support the IR lifecycle. Their study defines 18 core IR concepts—such as observables, indicators, Tactics, Techniques, and Procedures (TTP), response actions, and threat actors—and evaluates six prominent incident-response–focused formats across criteria including expressiveness, interoperability, and machine-readability.

A key finding is that all formats consistently emphasize the *Action* concept, reflecting the inherently action-oriented nature of incident response. However, no single format covers all concepts comprehensively. In practice, organizations often combine multiple CTI formats or frameworks to adequately address diverse operational use cases.

Beyond CTI-specific formats, the study also surveys several IR standards and frameworks. Among those discussed are CACAO, Collaborative Open Playbook Standard (COPS), Integrated Adaptive Cyber Defense (IACD), and Open Command and Control (OPENC2). The following subsections summarize visual results extracted from Schlette et al.'s comparative analysis.

2.7.1 Focal Points of Use Case Scenarios

The figure *Focal Points of Use Case Scenarios* illustrates how different use cases prioritize specific IR concepts. In the automation scenario, all structural concepts are mandatory: detailed workflow specifications, explicit actuator modeling, precise action descriptions, and integration of CTI artifacts. Technical requirements (machine-readability, standardized serialization) and authorization are also emphasized. In contrast, the sharing use case prioritizes confidentiality, sensitivity markings, playbook aggregability, and versioning. These results demonstrate that operational context significantly influences which concepts are most critical: automation stresses process detail and technical integration, whereas sharing highlights privacy and semantic clarity.

Table 2.2: Focal Points of evaluated metrics in use cases [10].

Concept \ Use Case	Automation	Sharing	Reporting
Aggregability	∘	++	+
Categorization	∘	+	++
Granularity	+	+	∘
Versioning	∘	++	∘
Referencing	+	+	+
Extensibility	∘	+	++
Readability	++	+	++
Unambiguous Semantics	+	++	+
Workflow	++	++	+
Actuator	++	∘	∘
Action	++	++	++
Artifact	++	+	∘
Community	+	++	∘
Application	++	+	∘
Serialization	++	++	∘
Confidentiality	∘	++	+
Authorization	++	∘	∘
Prioritization	+	+	∘

Legend: ∘ less relevant + supporting ++ mandatory

2.7.2 Structural Concept Implementation

The figure *Comparison of Structural Concept Implementation* shows that all evaluated formats strongly support the *Action* concept, which is expected since IR fundamentally involves executing response measures. Coverage gaps, however, are evident: CACAO provides only a weak artifact/CTI-integration element (lacking a dedicated Artifact concept), COPS has no explicit Actuator concept (depending instead on external services), and IACD specifies neither actuator nor artifact.

Table 2.3: Comparison of Structural Concept Implementation [10].

Format	Workflow	Actuator	Action	Artifact
CACAO	✓	✓	✓	×
COPS	✓	×	✓	✓
IACD	✓	×	✓	✓
OpenC2	×	✓	✓	✓
RE&CT	✓	×	✓	×
RECAST	×	×	✓	×

2.7.3 Technological Concept Implementation

The figure *Comparison of Technological Concept Implementation* compares the ability of different formats to model governance-related aspects such as confidentiality, authorization, and prioritization, together with the inclusion of an explicit security model. Among the evaluated approaches, CACAO is the only one that integrates all three properties and embeds security as a first-class concern. Requirements for the Expression of Cybersecurity Techniques (RE&CT) also provides confidentiality and authorization with an included security layer, but lacks explicit prioritization. Other formats, such as COPS, IACD, OPENC2 and Resilient Event Conditions Action System against Threats (RECAST), cover these dimensions only partially or omit them entirely, requiring external mechanisms to enforce governance. This highlights CACAO 's relative maturity, since governance features are essential in IR for ensuring controlled automation.

Table 2.4: Comparison of Technological Concept Implementation [10].

Format	Confidentiality	Authorization	Prioritization	Security
CACAO	✓	✓	✓	included
COPS	×	×	×	excluded
IACD	×	✓	×	excluded
OpenC2	×	×	×	excluded
RE&CT	✓	✓	✓	included
RECAST	×	✓	×	excluded

2.7.4 Specification, Literature, and Status

The figure *Comparison of Specification, Literature and Status* reports on the documentation and maturity of each format. CACAO is noted to have limited academic literature and few example implementations. COPS 's specification is sparse and the format is essentially inactive. IACD provides a moderately detailed playbook specification, albeit with narrow scope. Both RE&CT and RECAST perform poorly: RE&CT lacks formal schemas or structured documentation, while RECAST has been briefly described in a single paper and is no longer active.

Table 2.5: Comparison of Specification, Literature and Status [10].

Format	Specification (Detail)	Literature	Status
CACAO	[37] (high)	limited	active
COPS	[38] (low)	limited	inactive
IACD	[39] (medium)	available	active
OpenC2	[40], [117], [118] (high)	available	active
RE&CT	[41] (low)	none	active
RECAST	[42] (low)	none	inactive

2.8 Purpose and Structure of CACAO

The **CACAO** model is an open standard developed by the Organization for the Advancement of Structured Information Standards (OASIS) consortium with the goal of defining an interoperable format for representing security playbooks [11].

The primary purpose of CACAO is to facilitate the sharing and automation of responses to cybersecurity incidents through a structured, human-readable, and platform-agnostic representation. This enables greater consistency in the execution of security actions and easier integration between heterogeneous tools and teams.

CACAO supports the modeling of playbooks that include a series of executable steps, each of which can represent an automated action, a human decision, a logical condition, or a branching point in the flow. The structure of a CACAO playbook (Figure 2.3) includes several main elements:

- **Metadata**: descriptive information such as the name, version, author, and description of the playbook.
- **Playbook Workflow**: the execution flow composed of interconnected nodes (workflow steps) that define the order and conditions of execution.
- **Playbook Actions**: specific actions to be executed, which may include commands, API calls, or interactions with external systems.
- **Data Markings and Extensions**: mechanisms for managing the sensitivity of information and extending the standard.

2.8.1 Types of Workflow Steps in CACAO

The CACAO playbook model defines several types of workflow steps to represent the flow of execution and decision-making within a playbook. Each step defines a specific behavior, and together they allow for both linear and complex branching logic in response workflows.

The main types of steps are:

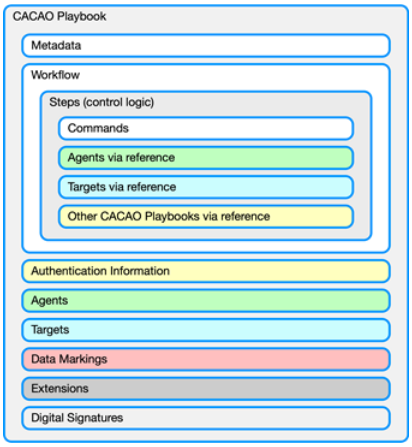

Figure 2.3: CACAO structure [11].

- **Action Step**: represents an executable action, such as invoking an Application Programming Interface (API) or interacting with an external system. It is the most important type and forms the core of playbooks.
- **If Condition Step**: introduces conditional branching based on a condition. It evaluates a condition and directs the workflow to different paths depending on whether the condition is true or false. Useful for implementing decision logic in response scenarios.
- **While Condition Step**: similar to a programming construct, it repeatedly executes a set of child steps as long as the defined condition holds true.
- **Parallel Step**: enables the concurrent execution of multiple child steps. It is particularly useful when independent actions can be performed simultaneously, improving overall response efficiency and reducing execution time.
- **Switch Step**: allows for multi-branch conditional logic based on a specified expression. It is functionally similar to a `switch-case` statement in traditional programming.

These step types, when combined, allow CACAO playbooks to describe not only linear sequences but also complex, dynamic workflows capable of adapting to varying incident conditions.

STIX Pattern Matching

Structured Threat Information Expression (STIX) is a standardized language developed by OASIS for representing cyber threat and observable information in a structured and machine-readable format [12]. Within the STIX 2.1 specification, the `pattern` object type is used to define observable conditions that indicate suspicious or malicious activity, using a dedicated query language known as the STIX Patterning Language (SPL).

In the context of CACAO playbooks, STIX patterns are primarily employed within conditional steps, such as `if-step` or `while` steps, to express decision logic based on the presence of indicators of compromise IoC or specific threat behaviors observed in the environment. These conditions allow a playbook to adapt its execution path based on contextual threat data.

2.8.2 Agents and Targets in CACAO

In a CACAO playbook, each executable action is associated with two fundamental components: an **agent**, which performs the action, and a **target**, which is the entity upon which the action is executed. This abstraction provides a flexible mechanism to decouple the definition of an operation from the specific actors involved, enabling reusable and context-sensitive playbook logic. Agents and targets are represented as entries in dedicated dictionaries, where each entry maps an identifier to an `agent-target` object. These objects share a set of common properties as defined in the standard, and they support a wide range of types and configurations [11].

Agents

An agent is the entity responsible for executing a command. Agents may operate automatically, such as a firewall rule engine or an orchestration platform, or manually, when an action requires human intervention. The agent type vocabulary includes various categories, such as:

- Security devices (e.g. firewalls, intrusion prevention systems);
- Networking infrastructure (e.g. routers, switches);
- Threat intelligence platforms;
- Human operators;

Agents can be dynamically configured using variable substitution. Instead of hardcoding values such as names or Internet Protocol (IP) addresses, variables can be used to inject runtime information into the agent configuration, making the playbook more adaptable.

Target

Targets are the recipients of the actions defined in a playbook step. Much like agents, targets can be systems, services, or individuals, and they are defined using the same structural model. Common examples include:

- IP addresses, domains, and Media Access Control (MAC) addresses;
- Uniform Resource Locator (URL)s or endpoints exposed by external systems;
- User accounts or identities (UID);

Each target may include metadata describing its nature and location, and it may also leverage variable substitution for runtime customization. Workflow steps that perform actions typically reference both an agent and a target by their identifiers. This mapping allows for clear traceability of who performs what operation and on which object.

2.8.3 Data Markings and Signature Support in CACAO

The CACAO 2.0 standard provides robust mechanisms for ensuring the confidentiality, integrity, and trustworthiness of playbook content. These mechanisms are essential for enabling secure sharing, ensuring compliance with data handling policies, and validating the origin and authenticity of shared artifacts.

Data Markings

Data markings in CACAO playbooks allow producers to specify handling instructions and sensitivity levels for individual elements or entire playbooks. For example, playbooks may be shared with the restriction that they must not be re-shared or that they must be encrypted at rest. These markings are defined using the `marking-definition` object, which supports both predefined and custom marking structures.

Markings are applied to specific objects using the `object_marking_refs` field, which references one or more `marking-definition` objects. Markings enable fine-grained control over how different parts of a playbook must be handled or shared.

Digital Signature Support

The CACAO v2.0 standard provides a standardized structure for digitally signing playbooks through the optional signature object [11]. This object allows producers to assert the authenticity and integrity of a playbook by attaching a digital signature using recognized cryptographic algorithms. This signature mechanism is designed to support external validation of playbook content, enhancing trust and traceability in automated security workflows. Multiple signature objects may be included in a playbook to support layered trust models (e.g., organization-level and individual author signatures).

2.9 MITRE ATT&CK Framework

One of the most widely adopted standards in the cybersecurity community is the MITRE Adversarial Tactics, Techniques, and Common Knowledge (MITRE ATT&CK) framework [13]. ATT&CK is a structured knowledge base that catalogues adversarial behaviours observed in real-world intrusions. It organizes malicious activities into a matrix of *tactics*, representing the adversary's technical objectives (such as persistence, privilege escalation, or lateral movement), and *techniques*, which describe the specific ways in which these objectives can be achieved (e.g., creating a new service for persistence or exploiting credential dumping for privilege escalation). Each technique is further enriched with information about mitigations, detection opportunities, and references to documented threat reports.

The value of ATT&CK lies in providing a common vocabulary and taxonomy to describe and reason about attacker behaviour. It enables defenders, vendors, and researchers to align on a shared language when discussing threats, designing detections, or evaluating coverage. By abstracting from specific malware or tools, ATT&CK captures adversarial intent in a reusable form, which makes it suitable both for high-level threat modelling and for operational tasks such as IR.

Chapter 3

Related Works

This chapter presents the main works, technologies, and standards that provide the foundation and context for the development of the tool introduced in this thesis. The discussion begins with the *Palantir Incident Response Tool*, which served as the starting point for this work, before examining additional components, such as the *iptables* engine, which supports remediation capabilities.

A substantial part of the chapter is dedicated to the CACAO standard, which plays a central role in shaping the design of the proposed tool and ensuring compliance with widely recognized models for playbook-based automation.

3.1 Palantir Incident Response Tool

The Palantir project [14, 15] included an automated incident-response engine, called the Recommendation and Remediation (RR) component, which processed security alerts and executed predefined remediation playbooks. When an alert is received, the RR tool first classifies the threat (e.g., botnet, brute-force attack) and uses a priority table to select an appropriate course of action. This action is encoded as a high-level "recipe" or playbook in a custom policy language. The system generates a file (with a Recovery file format (REC) extension) containing the chosen playbook, then invokes a grammar-based interpreter to execute it. In other words, the RR engine translates the abstract recipe into concrete commands that reconfigure the network or services. These commands may modify the virtual service graph (for example, by deploying or adjusting virtual firewall/Virtual Network Function (VNF) nodes) or issue low-level device instructions (such as adding or removing `iptables` rules) to contain the threat. Throughout this process, the system updates a dashboard to inform the operator of detection and remediation events.

For the actual enforcement of remediation actions, the RR component relies on an orchestration layer. Once a recipe has been parsed and mapped to concrete commands, the tool interacts with the Management and Orchestration (MANO) API to deploy new security capabilities (e.g. intrusion detection nodes, honeypots, or filtering firewalls) and to reconfigure existing ones.

3.1.1 Inherited Components from the Palantir Tool

Although the core logic of the incident-response engine has been significantly revised in this work, several key components of the original Palantir RR tool have been preserved and adapted. The main contribution of this thesis is the definition of an executable instance of a playbook that can represent a remediation procedure, designed to be compliant with the CACAO standard and enriched with cloud-specific functionalities. Nevertheless, continuity with the original implementation was maintained in the following aspects:

- **Recipe Compatibility.** The system still accepts remediation instructions encoded in REC files and interprets them using the original `grammar.tx`. However, while the grammar remains unchanged, the internal logic that maps recipes into executable playbook instances has been redefined to comply with the CACAO model and to support different remediation actions.
- **Service Graph.** The service graph abstraction, used to represent the infrastructure and the relations among nodes, has been largely preserved. Although the configuration of nodes has been revised to reflect new deployment scenarios, the graph operations (e.g., insertion, removal, and reconfiguration of security functions) follow the same logic as in the Palantir tool.
- **Alert-to-Response Mapping.** The strategy for linking incoming alerts to remediation procedures has evolved. In this work, the remediation process primarily relies on the MITRE ATT&CK/MITRE Defensive Engagement Framework (MITRE DEFEND) knowledge bases to identify the most suitable response. However, if an alert does not include a MITRE technique identifier and only specifies a more general *threat category*, the original mapping logic from the Palantir tool is reused. This approach ensures backward compatibility of alerts and provides a fallback mechanism for less-structured alerts.

3.2 Formal model of capabilities of Network Security Functions

A relevant component employed in this thesis is the IP Tables Firewall (iptables) engine, developed at Politecnico di Torino. This work [16] presents a formal capability-based model for automatically generating firewall rules from abstract security policies. The goal is to bridge the gap between high-level policy specification and low-level enforcement, reducing the dependency on vendor-specific configuration languages.

At the core of the engine lies the concept of security capabilities, which formally describe what a Network Security Function (NSF) can do in terms of traffic selection and actions. Capabilities are organized in six categories:

- Conditions to select the traffic of interest (e.g., IP addresses, ports, protocols).
- Actions that can be applied to that traffic (e.g., accept, deny, encrypt).

- Events that may trigger rule evaluation.
- Condition clauses that specify whether rules require conjunctive or disjunctive matching.
- Resolution strategies for handling multiple matching rules.
- Default actions when no rule applies.

From a technical perspective, the engine adopts a model-driven approach. An abstract configuration language is automatically generated from the description of an NSF's capabilities. Then, using an adapter pattern, these abstract configurations are translated into the concrete syntax of the target NSF. For iptables, this means that each abstract capability is associated with the corresponding low-level option through an adapter. For instance, the abstract capability "SourcePortCondition" is translated into the concrete option `-sport`, while "AppendRuleCapability" becomes the `-A` flag used to add a rule to a chain. The adapter also encodes the dependencies among the options and provides the syntactic rules required to construct valid firewall commands.

An important role is played by Extensible Markup Language (XML), which is used to represent the capability model and its instantiations. Capabilities, abstract languages, and translation rules are expressed through XML Schema definitions, enabling the use of standard XML tools for validation and manipulation.

The validation of the engine on iptables showed that abstract policies, expressed using the generated language, can be consistently and correctly translated into working firewall rules. In practice, an abstract rule such as "accept TCP traffic from port 80 to subnet 192.168.1.0/24" is automatically converted into the command:

```
iptables -A FORWARD -p tcp --sport 80 -d 192.168.1.0/24 -j ACCEPT
```

3.3 CACAO-Compliant Tools as References

This section describes the CACAO-compliant tools employed during the development process, with a focus on their functionalities and roles in the implementation and validation phases.

3.3.1 SOARCA

Security Orchestrator for Advanced Response to Cyber Attacks (SOARCA) is an open-source security orchestration platform designed to ingest, validate, and execute CACAO v2.0 playbooks via a JSON API [17]. SOARCA provides a fully CACAO-compliant execution engine aimed at experimental use cases and the orchestration of automated remediation in security operations.

From a technical perspective, SOARCA acts as an intermediary layer between CACAO playbooks and the execution environment. Once a playbook is submitted, the system parses the JavaScript Object Notation (JSON) representation, validates its schema against

CACAO specifications, and instantiates the corresponding remediation classes. Each playbook step is then mapped to an actionable task, which is dispatched to the appropriate connector or protocol handler e.g., Secure Shell (SSH) for host-level actions, HyperText Transfer Protocol (HTTP) for API calls, or Message Queuing Telemetry Transport (MQTT) for Internet of Things (IoT) oriented responses. The orchestration logic ensures that conditional flows, dependencies, defined in CACAO are respected during runtime.

During the initial stages of the project, SOARCA was examined to understand the instantiation process of remediation classes from JSON-formatted CACAO playbooks. The examples available in the official repository were used as a practical reference to analyze how different step types are structured and integrated into the overall playbook workflow. This analysis was particularly valuable given that the examples provided in the official CACAO documentation are often partial.

3.3.2 CACAO Roaster

CACAO Roaster is a web-based application maintained by the Open Cybersecurity Alliance (OCA) for the creation, validation, visualization, and execution of CACAO v2.0 playbooks [18]. It provides a user-friendly, no-code interface that supports both manual design, with drag-and-drop, and schema validation of playbooks in accordance with the OASIS CACAO specification.

From a technical standpoint, CACAO Roaster abstracts the complexity of the CACAO JSON schema into a graphical editing environment, where each playbook element (such as actions, conditions) can be instantiated and connected visually. Behind the interface, the tool automatically generates a standards-compliant JSON representation, ensuring that the resulting playbooks can be directly interpreted by execution engines such as SOARCA. In addition to design and visualization, Roaster integrates validation routines that check playbooks against the official CACAO schema, highlighting structural inconsistencies or missing fields before deployment.

During the development process, CACAO Roaster was extensively utilized for the validation of playbooks generated by the tool described in this thesis. It proved valuable for verifying schema correctness, reviewing workflow structure, and testing conformance with CACAO specifications. Its integration capabilities and graphical representation helped ensure that the generated playbooks were both syntactically correct and operationally valid.

Chapter 4

Problem Statement

This chapter outlines some of the key challenges that remain unsolved in the current landscape of incident response automation, which this thesis seeks to address. Specifically, it highlights the limitations of the CACAO standard in covering dynamic scenarios, as well as the lack of mechanisms for sharing concrete remediation executions. Finally, it discusses the lack of a dedicated Kubernetes remediator, despite the growing adoption of containerized and cloud-native infrastructures.

4.1 Limitations of the CACAO Standard

As mentioned above, CACAO offers a rich set of workflow constructs – including sequential steps, parallel branches, conditionals, and while-loops – that give organizations flexibility in encoding complex response logic. Despite CACAO's features, existing implementations reveal gaps in some scenarios.

CACAO's standard syntax enforces certain constraints that limit expressiveness in dynamic playbooks. Conditions in CACAO are typically specified as STIX pattern strings, meaning they compare constant fields against constant values. However, there is no native mechanism for conditions that bind to outputs of earlier steps (such as binding a variable from a previous action into the next condition). As a result, any branching logic that depends on runtime data must be handled outside the standard playbook flow or via custom extensions.

Similarly, looping constructs in CACAO are limited. The specification lists only while-loops as the repetition primitive. No explicit syntax exists for iterating over all elements in a list or collection. Thus, use cases such as "apply this remediation to each container/pod found" cannot be expressed directly. In practice, this means CACAO playbooks lack support for "foreach" or output-based loops and require workarounds.

4.2 Playbook Sharing and Execution Data

In many cybersecurity communities, sharing knowledge is crucial. Standards like STIX or Trusted Automated eXchange of Indicator Information (TAXII) enable sharing threat intelligence (indicators, TTP, etc.). For example, CACAO playbooks can be linked to STIX `CourseOfAction` objects or use CTI within conditions. However, the state of sharing today focuses on distributing playbook definitions or recommended actions, not the details of what happened during execution.

Interoperable security playbooks should be part of structured information sharing [19]. This approach standardizes the format of playbooks for exchange. What it does not capture, however, is the actual runtime data from executing those playbooks. In other words, there is no standard way to share a playbook *with its execution transcript*: the inputs given to each function and the outputs produced. Such enriched playbooks would be very useful – akin to sharing not just a recipe, but the actual results and data from cooking it. Notably, CACAO's schema allows parameters to be marked as internal (computed during execution) versus external, and it is extensible, which could support the inclusion of execution-state data in the future.

4.3 Kubernetes and Remediation Tools

Container orchestration platforms, such as Kubernetes, are now ubiquitous in production infrastructure. Nearly all large organizations have adopted Kubernetes, and its use continues to climb. Despite this prevalence, existing incident response frameworks rarely provide out-of-the-box Kubernetes-specific remediation. In the context of CACAO, there is no widely adopted execution engine that specializes in Kubernetes actions. In other words, a "playbook-driven remediator" for Kubernetes (to automatically apply playbook steps to pods, network policies, services, etc.) is essentially lacking.

4.3.1 Identified Gap

From the discussion above, several functionalities are missing in the current incident response landscape. This thesis aims to address the following gaps, outlined in the list below.

- **End-to-end automation pipeline:** A workflow that ingests enriched security alerts and produces executable actions tailored to a specific environment.
- **Dynamic instance generation:** Support for playbooks where inputs (variables or targets) are computed at run time, rather than being statically defined.
- **Round-trip transformation:** The ability to translate executed flows back into CACAO format. For example, once a custom automation (possibly coded outside CACAO) has been executed, it should be possible to export the actual action sequence and data back into a valid CACAO playbook for documentation or sharing.

- **Cross-environment execution:** A remediation backend capable of operating seamlessly across cloud-native platforms (e.g., Kubernetes APIs, Cilium policies) and traditional infrastructures (e.g., iptables, on-prem devices, graph-based network models).
- **Rich control flow at runtime:** Full support in the execution engine for advanced logic such as dynamic condition evaluation on action outputs, branching on results, parallel execution of steps, and non-trivial loops (e.g., `foreach` over variable-length lists).

Chapter 5

Design

5.1 Design Objectives

The primary goal of the proposed tool is to support and automate *incident response* activities by providing a flexible and extensible framework for the execution of defensive actions. The cornerstone of the design is the implementation of a Python class capable of representing an *executable instance* of a playbook. This abstraction allows the system to manage the lifecycle of a defensive action from its selection to its execution and, optionally, its generation in standardized formats. As shown in Figure 5.1, the main components and processes that constitute the tool are presented below.

- **Alert-to-Recipe Mapping:** Given an incoming security alert, the tool must identify and map it to the most appropriate defensive recipe (e.g., a `.rec` file or a CACAO playbook) to ensure an effective and timely response.
- **Recipe Parsing and Instantiation:** The tool must be able to parse an input recipe, regardless of whether it is defined in a proprietary format (`.rec`) or follows the CACAO standard, and convert it into an executable Python object instance.
- **Playbook Execution:** Once instantiated, the playbook instance must be executable, orchestrating the defined defensive actions according to the prescribed logic and conditions.
- **CACAO Playbook Generation:** The tool should also provide the capability to generate a valid CACAO-compliant playbook starting from an executable instance, enabling interoperability with other incident response platforms.

5.2 Formats Used in the Tool

The tool developed as part of this thesis integrates and supports multiple data representation formats, each chosen for its suitability to a specific aspect of the security automation workflow. The most relevant formats are JSON, YAML Ain't Markup Language (YAML).

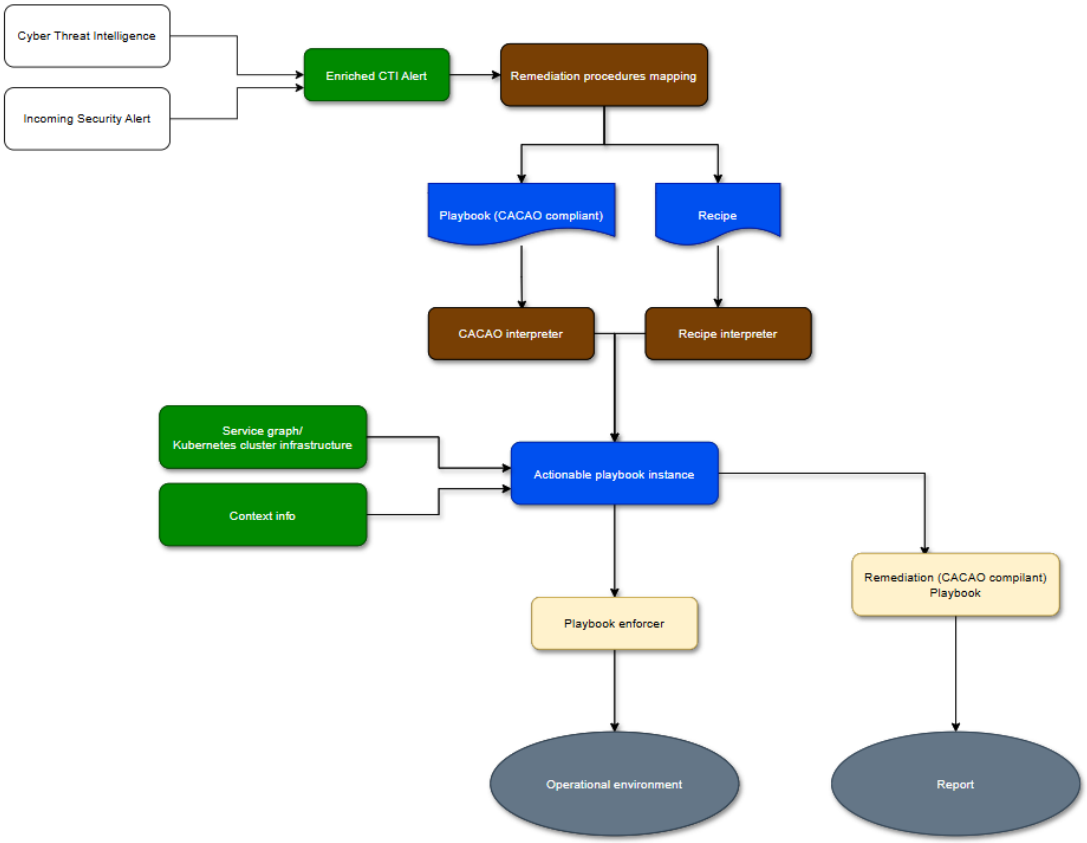

Figure 5.1: PB-rem scheme

5.2.1 JSON

JSON is the core format used for handling both the representation of playbooks and the management of alerts. Within the tool, CACAO playbooks are instantiated and executed in JSON format, ensuring compatibility with the standard defined by the CACAO specification. [11].

In addition to generating executable instances from CACAO playbooks, the tool also supports the reverse process: transforming an arbitrary executable playbook (not initially conforming to CACAO) into a valid CACAO representation. This bidirectional capability allows for flexible integration and standardization.

Furthermore, alerts received and processed by the tool are internally represented using JSON objects (Python dictionaries), providing a lightweight and efficient structure for communication between system components.

5.2.2 YAML

YAML is used primarily in the context of cloud security policies. The tool supports the generation of policy definitions in YAML through the use of templates. Specifically, it integrates with two policy enforcement frameworks: **KubeArmor** and **Cilium**.

In both cases, the tool fills predefined YAML templates with the relevant security data discovered during analysis. These YAML files are then applied to enforce runtime security at the Kubernetes level, enabling isolation or remediation actions.

5.3 Tool components

In this section, we will list the high-level structures implemented for the features just shown.

5.3.1 Ingestion Layer

The *Ingestion Layer* is responsible for receiving incoming alerts and extracting the information required for further processing. A key design choice is to map each enriched alert to a standardized data structure based on its `threat_category`. This mapping is defined in a global scope through a JSON configuration file, which serves as a reference for identifying and normalizing the relevant fields for each category.

The configuration file follows the structure shown below:

```
"name_threat_category": {
    "mandatory_field": {},
    "context_info": {},
    "optional_fields": {}
}
```

The entries in the configuration file are defined as follows:

- **mandatory_field**: Contains the essential fields required to execute any remediation action for an alert belonging to the given `threat_category`. Examples include the IP address of an impacted host or the address of an attacker host. These fields are considered mandatory for the remediation process to proceed.
- **context_info**: Specifies the additional information needed to perform the remediation in the given context. For example, in a cloud environment, it may include cluster certificates or API credentials necessary to apply defensive actions.
- **optional_fields**: Lists fields that are not common to all alerts of the given `threat_category`, but can be required to execute the remediation process for certain alerts within that category. Examples include the infected node in a cluster for node-level alerts or the specific pod name for pod-level alerts.

It is essential to note that this configuration file associates a `threat_category` with a class of alerts that share the same remediation requirements, so it does not describe a specific alert instance.

5.3.2 Alert-to-Defensive Action Mapping

The mapping of an alert to the corresponding defensive actions is performed according to one of two strategies, depending on the information available in the alert.

Mapping by MITRE Code

When the alert contains a `mitre_code` entry, the mapping process (Figure 5.2) follows a multi-step resolution path based on four JSON configuration files:

1. **MITRE ATT&CK Techniques Repository:**

   ```
   {
       "id": "mitre_code",
       "name": "",
       "description": "",
       "defensiveCategories": []
   }
   ```

   This file links each `mitre_code` to one or more `defensiveCategories`.

2. **Defensive Categories Repository:**

   ```
   {
       "name": "",
       "defensiveTechniques": []
   }
   ```

   Each `defensiveCategory` is associated with a set of `defensiveTechniques`.

3. **Defensive Techniques Repository:**

   ```
   {
       "name": "",
       "description": "",
       "defensiveActions": []
   }
   ```

   Each `defensiveTechnique` leads to one or more `defensiveActions`.

4. **Defensive Actions Repository:**

```
{
    "id": "DA_id",
    "description": "",
    "playbook_name": "",
    "playbook_type": "",
    "RequiredParameters": []
}
```

This file contains the executable `playbook` associated with a given defensive action. The `playbook_type` field specifies whether the playbook is in CACAO format or a recipe.rec. The `RequiredParameters` list specifies the parameters that must be present in the `global_scope` for the defensive action to be executed successfully. This entry is useful in case one of the optional parameters is needed to execute that specific remediation.

Mapping by Threat Label

If the alert does not contain a `mitre_code`, the mapping is performed using the threat-label field. In this case, recipes are associated with each `threat_category` and `threat_label` according to a priority-based mechanism, defined in the `threat.json` file:

```
"threatsrepository": {
    "threat_category": {
        "threat_label": {
            "recipes": [
                { "recipeName": "", "priority": ,"impact": },
                    ...
            ]
        },
          ...
    },
     ....
}
```

Each recipe entry contains:

- `recipeName`: the identifier of the recipe to execute,
- `priority`: a numeric value used to rank recipes when multiple are available,
- `impact`: a numeric estimated impact score of the recipe.

The priority value ensures that, in cases where multiple recipes are applicable, the system consistently selects the most appropriate defensive action for the given threat.

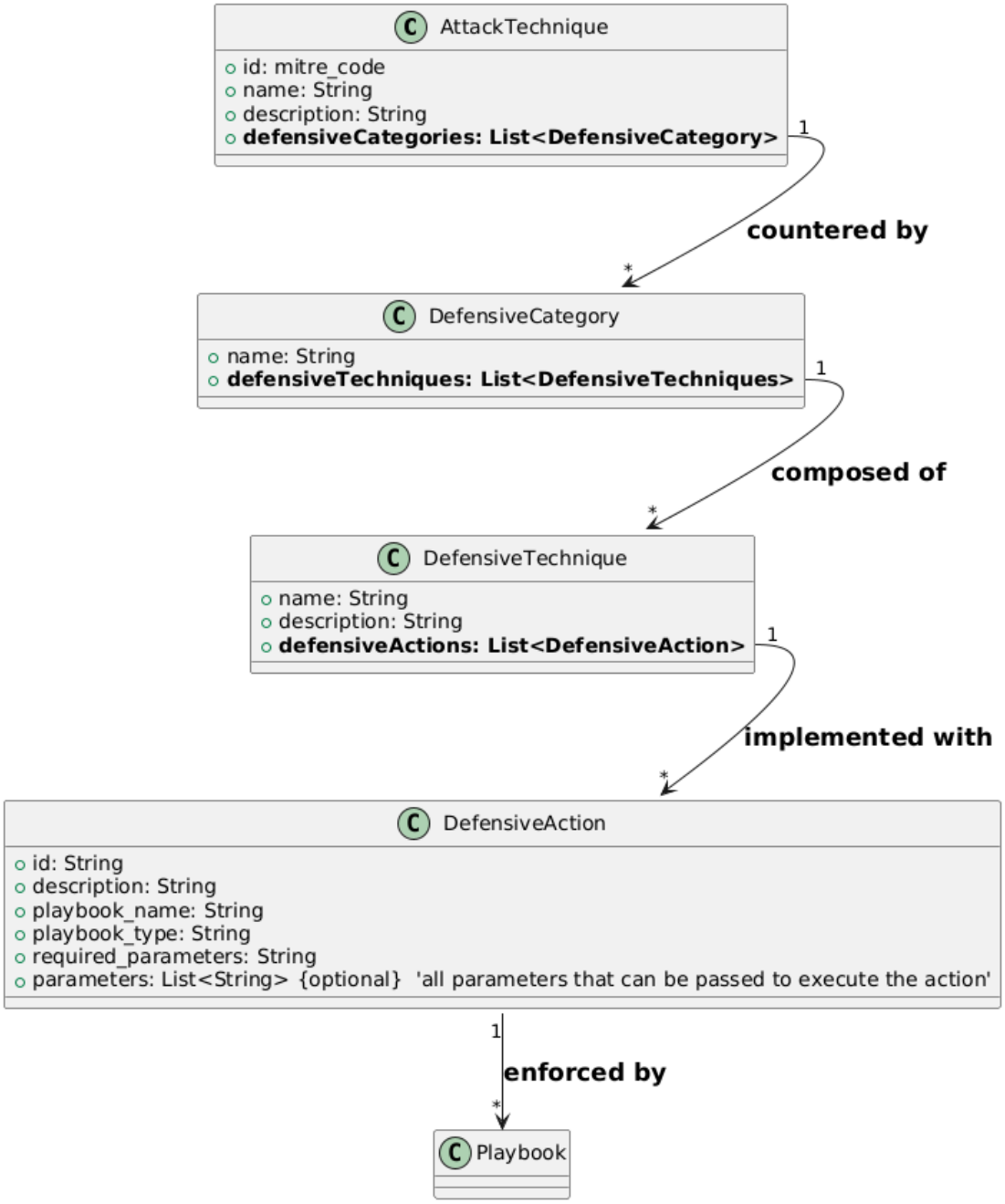

Figure 5.2: Threat intelligence configuration

5.4 Playbook Class Design

One of the most critical components of the system is the `Playbook` class. This section focuses on its design principles without delving into low-level implementation details. The `Playbook` class encapsulates all the data and logic required to orchestrate the execution of defensive actions in a structured and modular way.

5.4.1 Core Attributes

The `Playbook` class maintains a set of parameters that are essential for execution, including:

- An instance of the `Executor` class, which is responsible for managing and executing the actions defined in the playbook.
- A graph representation of the execution nodes, modelling the relationships between steps.
- The `global_scope`, which stores contextual and operational data required for execution.
- A list of `Step` objects representing the ordered execution flow.

5.4.2 Step Abstraction

Each step is represented by an abstract `Step` class, which is inherited by more specific step types, such as:

- `ActionStep`
- `LoopStep`
- `ConditionStep`
- `ParallelStep`

All step types share two fundamental attributes:

1. A human-readable name.
2. A Universally Unique Identifier (UIID)-based identifier, serving two purposes:
    - Identifying the step in error or debug messages.
    - Linking steps during CACAO playbook generation from the executable instance.

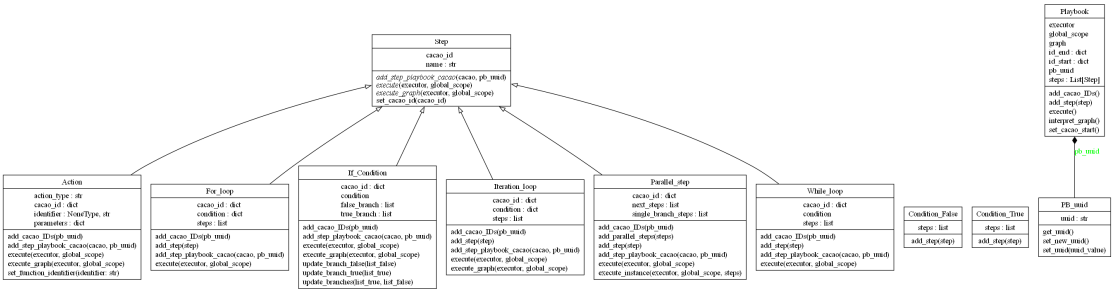

Figure 5.3: Graphviz playbook structure

5.4.3 Key Methods

Most step types also share a common set of methods. The most relevant is `execute`, which defines the behaviour of the step at runtime. In certain cases, such as when processing graph structures, a method `execute_graph` may also be provided.

The `add_step` method is used within the `Playbook` class to extend the execution flow by adding new steps. Additionally, it is also employed by specific step instances, particularly for non-action steps, where it enables the definition of their internal execution logic. In these cases, the method is used to:

- Define the `true` and `false` branches in conditional steps.
- Specify the parallel execution branches in a `ParallelStep`.
- Attach the sequence of actions to be repeated within a `LoopStep`.

5.4.4 Class Structure Representation

The `Graphviz` diagram (Figure 5.3) illustrates the structural relationships of the `Playbook` object and its associated step types:

5.5 Condition design

This section will discuss solutions designed to handle conditions (useful for both loops and conditional structures) in remediation.

5.5.1 Condition Handling

As a design choice, the tool supports three distinct types of conditions, each serving a specific purpose in the execution flow.

Firstly, to ensure CACAO compliance, the tool is capable of interpreting STIX patterns directly, allowing standardised conditional logic as defined in the CACAO specification.

Secondly, the tool supports the condition syntax used in `.rec` files. These are of the form:

```
if [not] variable_name
```

In this case, the condition evaluates whether the value of the given variable, retrieved from the `global_scope`, is considered `true`. The optional `not` keyword allows the logical negation of the condition.

Finally, a custom condition format is provided to represent scenarios that cannot be expressed using STIX patterns (as outlined in the Problem Statement). The format is as follows:

```
"condition": {"lhs": var1, "op": operator, "rhs": var2_or_value}
```

Here:

- `lhs` (*left-hand side*) – the first operand, the name of a variable or referencing data from the `global_scope`.
- `rhs` (*right-hand side*) – the second operand, which may be either a literal value or the name of a variable in the `global_scope`.
- `op` – a dictionary entry mapping a string symbol to a Python operator from the `operator` library:

An example of such a condition is:

```
"condition": {"lhs": "vulnerable_pods['length']", "op": ">=", "rhs": 1}
```

This evaluates to `true` if the attribute `length` of the `vulnerable_pods` dictionary in the `global_scope` is greater than or equal to `1`.

The system also supports something like:

```
"condition": {"lhs": "vulnerable_pods['length']",
              "op": ">=",
              "rhs": var(threshold)}
```

If the `rhs` value is not a hard-coded value, the tool attempts to retrieve a variable with that name from the `global_scope`.

5.6 Execution Environments

Each of the classes above implements its own `execute` method, tailored to the specific purpose of the class. While the detailed behaviour of these methods will be discussed in the *Implementation* chapter, it is important to note here that they define the runtime execution logic of the playbook and its individual steps.

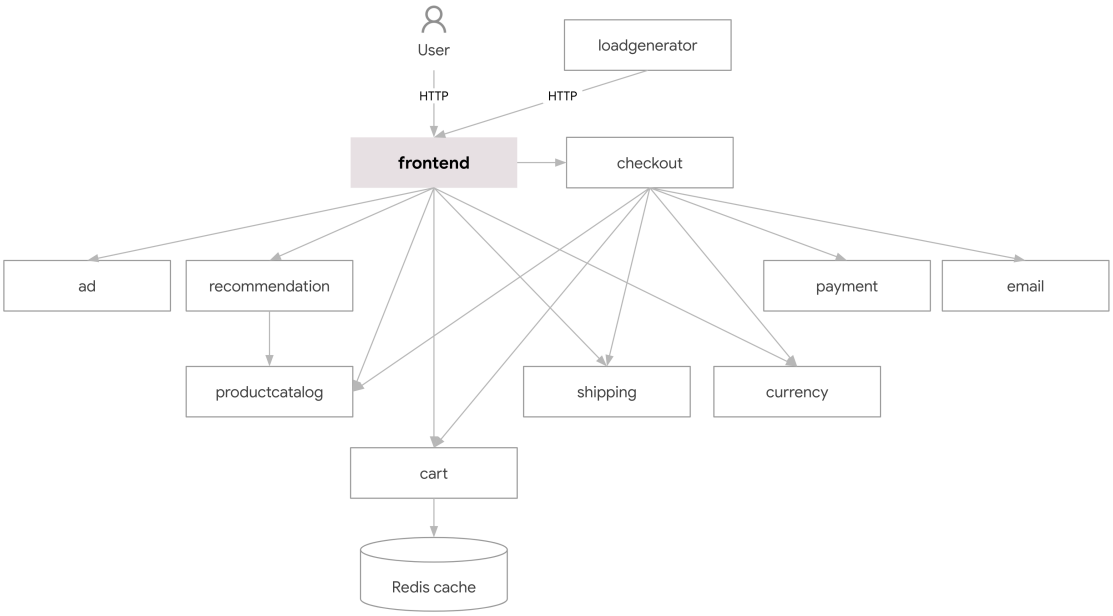

Figure 5.4: Cluster configuration [20]

5.6.1 Cloud Environment

The cloud execution environment is tested using the Python Kubernetes API against a Google Kubernetes Engine (GKE) cluster running the `microservices-demo` application. In the figure 5.4 the configuration of the eleven microservices. The specifications and deployment instructions for this cluster can be found in the official repository:

`https://github.com/GoogleCloudPlatform/microservices-demo`

An optional appendix will include a detailed guide on how to interface this cluster with the developed tool.

Alongside the cluster, a configuration file named `kubernetes_infrastructure.json` is maintained. This file is currently used to manually mark malicious pods for testing purposes, but it is designed to support richer functionality in future iterations of the tool.

The file follows the structure shown below:

```
{
    "pods": [
        {
            "label": "frontend",
            "networkProperties": {
                "service": {
                    "protocol": "TCP",
                    "port": 8080
```

```
                },
                "restrictions": {
                    "enforcement_behavior": "",
                    "rules": []
                }
            },
            "libraries": [
                {
                    "name": "go",
                    "version": "1.23.4",
                    "vulnerabilities": [
                        {
                            "id": "CVE-2024-45336",
                            "cvss3.x": 6.1,
                            "description": ""
                        }
                    ]
                }
            ],
            "serviceDependencies": [
                {
                    "label": "checkoutservice"
                }
            ],
            "artifacts": [
                {
                    "id": "",
                    "name": "",
                    "definition": "",
                    "d3fendCountermeasures": [],
                    "att&ckTechniques": []
                }
            ]
        },
        ...
    ]
}
```


This structure allows the definition of pods along with their network properties, dependencies, libraries, and known vulnerabilities. In addition, it provides the ability to annotate MITRE ATT&CK techniques and MITRE DEFEND countermeasures associated with specific artifacts. While the current usage is limited to marking pods as malicious, future enhancements could leverage this configuration for more advanced threat modelling and automated defensive responses.

Kubernetes agents

Within the current setup, cloud-related actions are supported either through direct Kubernetes API interventions or by enforcing network policies via Cilium. While the tool can also generate KubeArmor policies, these are not applied in *enforcement* mode. This choice reflects the focus of the envisioned attack scenarios and corresponding playbooks, which primarily target lower-level actions. Consequently, KubeArmor policies remain at the stage of hypothesis and require manual validation before being deployed in production environments.

5.6.2 On-Premises Environment

The On-Premises Environment relies on a graph-based network configuration model implemented using the `igraph` library. In this model, each node of the graph represents a possible network component, such as firewalls, hosts, servers, or switches. This approach allows the simulation of complex network topologies and facilitates the testing of defensive actions in a controlled environment.

A visual representation of the graph can be generated and plotted to verify network properties or the placement of specific components. For example, the plot can confirm that a firewall has been correctly positioned between an attacker node and a victim node, as illustrated in the example below.

In addition to network topology simulation, this environment also supports a remediation mechanism for firewall configuration. Specifically, the tool can generate and output `iptables` rules by leveraging a Java ARchive (JAR)-based engine developed, as mentioned above, by the Politecnico di Torino in previous years as part of a capability model. [16].

Chapter 6

Implementation

This chapter presents the practical implementation of the system described in the design phase. While the previous chapter focused on the conceptual architecture and the logical structure of the components, here we detail how these concepts were translated into executable code and how they are deployed in real environments. The chapter covers the mechanisms used to parse and map CACAO playbooks into executable playbook instances, the concrete implementation of actions, and the end-to-end process from alert ingestion to the execution of defensive measures. Differences between cloud and on-premises execution environments are also discussed, together with the strategies adopted for error handling, logging, and monitoring during runtime execution.

6.1 Creating the Executable Playbook from CACAO

The code responsible for creating the executable playbook instance from a CACAO playbook is explicitly designed to avoid recursion. While a recursive approach can naturally reflect the hierarchical structure of the conditional and looping constructs of the playbook class, it does not reflect the cacao construct, which intentionally represents playbooks as a list of steps. Instead, the implementation relies entirely on vectors and queues to maintain a clear and controllable state during parsing.

At the heart of the process, a queue-like list (`self.pb`) is used to track the current insertion point in the playbook structure. The last element of this list is always the structure to which new steps will be added. Initially, this list contains only the newly instantiated `Playbook` object to be populated.

When a construct such as a `while` loop is encountered, a corresponding `While_loop` object is created and inserted into the list. From that point on, all subsequent steps will be added to this `while` object via its `add_step` method (common to all step types). Parallel to this operation, the `on_completion` UUID of the loop is stored in a dedicated list.

Subsequently, if a step with this UUID is reached, the algorithm will pop the last element from the playbook stack and add the completed `while` block to the previous structure, thus closing the scope of the loop without requiring explicit `end` markers.

A similar approach is applied to `if` conditions and `parallel` constructs, each handled via its own tracking lists (`on_completion_if`, `on_completion_parallel`, etc.). Each type of construction has its own completion list because a step could be the end of an if, a while, or a parallel step at the same time. This design choice ensures that the implementation works even though the CACAO playbook does not explicitly include `end` steps for these constructs, which is clearly not required by the official specification.

In summary, the algorithm processes the workflow sequentially, using:

- A primary stack (`self.pb`) to keep track of the current insertion context.
- Auxiliary completion lists to detect when a logical block should be closed.
- A unified `add_step` interface, shared by all construct types, to insert nested actions consistently.

Listing 6.1: CACAO Playbook parser

```
self.pb.append(playbook)
        for key, step in self.workflow.items():

            if (f"{self.on_completion_parallel[-1]}" in key):
                self.update_parallel()
            if (f"{self.on_completion_parallel_final[-1]}" in key):
                self.update_final_parallel()

            if (f"{self.on_completion_true[-1]}" in key):
                self.update_branch_true()

            if (f"{self.on_completion_if[-1]}" in key):
                self.update_branch_false_and_add_step_if()

            if (f"{self.on_completion[-1]}" in key):
                self.add_step_while()

            handler = self.handlers.get(step["type"])
            if handler:
                handler(step)
        return playbook
```

Notes on Parallel Steps Parallel step handling follows similar stack-based logic, but with additional considerations for managing multiple branches. When a parallel step is detected, the implementation adds the same instance of the `Parallel_step` class to `self.pb` as many times as there are entries in the `next_steps` list. In parallel, the `self.on_completion_parallel` list is extended with the corresponding UUIDs of the `on_completion` handlers for each branch.

Subsequent actions for each branch are temporarily stored in the `single_branch_steps` attribute of the `Parallel_step` instance. When the end of a branch is reached, these `single_branch_steps` are added to the branch list (`branches`) of the same `Parallel_step`

instance. Finally, following the same stack unwinding process described for other constructs, the last entries in both `self.pb` and `self.on_completion_parallel` are popped, effectively returning the flow to the previous execution context. Once the parallel step's on-completion is reached, the program behaves as described above, adding the last branch, removing the final self.pb from the stack, and adding the playbook class to the last element of self.pb.

Notes on Actions In the implemented system, playbook actions and remediation actions are both mapped to an internal `execution_function` code. While this attribute is not part of the official CACAO specification, it can be embedded in the `step_extensions` field as allowed by the standard's extensibility model. If such an `execution_function` is present, the action is refined by invoking:

```
action.set_function_identifier(execution_function_code)
```

6.2 Executable Playbook Generation in On-Premises Environments

In the on-premises configuration, executable playbooks are generated starting from textual *recipe* files (with `.rec` extension) written in a domain-specific language (DSL) specifically designed for remediation procedures. The DSL grammar, defined in `grammar.tx`, describes control structures (`if`/`else` conditions, `iterate_on` loops) and domain-specific remediation actions (e.g., `list_paths`, `add_firewall`, `shutdown`). Each grammar rule (action or control constructs) is bound to a corresponding Python class that implements the `run` method.

The interpreter (`textx_interpreter_playbook.Interpreter`) loads the DSL metamodel via `textX` from `grammar.tx`, parses the recipe, and constructs an in-memory model. The model's `run` method is then executed, receiving two inputs: the *global scope* (a shared state containing variables and context) and an initially empty playbook instance.

Listing 6.2: Interpreter class

```
@dataclass
class Interpreter:

    globalScope: object
    playbookFile: str

    def launch(self,playbook)->None:
        playbook_mm = textx.metamodel_from_file("source/
    recipe_interpreter/grammar.tx", classes=playbook_classes)
        playbook_m = playbook_mm.model_from_file(self.playbookFile)
        playbook_m.run(self.globalScope, playbook)
```

Each DSL element class encapsulates the logic required to map recipe instructions into executable playbook steps. For example, the `list_paths` instruction retrieves source and

destination parameters from the scope, creates an `Action` object with the appropriate type and parameters, and appends it to the playbook via `playbook.add_step()`. Control flow constructs in the recipe determine the order and conditional execution of these steps.

A key aspect of this execution model (contrary to what was described before) is its recursive nature: control constructs (`if`, `iterate_on`) do not directly insert primitive actions into the main playbook, but instead execute their nested statements by invoking their `run` methods in turn. For instance, an `if` statement creates a conditional node (`If_Condition`) with separate branches for the *true* and *false* cases. Each branch is filled by iterating over its statements, recursively calling `run` on each of them, so that nested conditions or loops are naturally supported. Similarly, an `iterate_on` construct generates an iteration node (`Iteration_loop`) and recursively executes its body statements for each iteration element.

This design ensures that the control flow described in the recipe is faithfully reproduced in the executable playbook, supporting arbitrary levels of nesting without additional orchestration logic. Through this approach, the generation of the executable playbook is entirely driven by the structure of the recipe: the interpreter processes the model sequentially, invoking each element's `run` method according to the recipe's control flow.

6.3 From Executable Playbook to Actual Remediation

Once the executable playbook instance has been generated, its execution is performed by invoking the `execute` method on its top-level steps. As in the last generation phase described, the execution model is inherently recursive: control constructs do not directly perform remediation actions, but rather invoke the `execute` method of the steps they contain, propagating the execution flow through nested structures. This approach ensures that the logical structure of the playbook whether it originates from a *recipe* or from a CACAO compliant description is preserved and executed exactly as specified.

6.3.1 Execution of Actions

Primitive *Action* steps represent the atomic units of remediation. Each action is associated with an `execution_function` identifier, which may be stored in the `step_extensions` field for CACAO compliant playbooks. At runtime, this identifier determines the exact remediation procedure to invoke. Once the corresponding function is selected, it is executed with the parameters previously resolved from the playbook's scope. This mechanism isolates the high-level description of the remediation from the low-level execution logic, enabling flexibility in mapping abstract steps to concrete procedures.

Listing 6.3: performing actions by first retrieving the function name

```
1        ...
2        action_function = executor.FunctionMappings[f"{id_func}"]
3        action_function(executor, self.parameters, global_scope)
4
```

6.3.2 Execution of Iterative Constructs

Iterative constructs execute their body steps repeatedly according to the type of loop defined in the playbook. While the general principle—iterating over a collection and invoking the contained steps is shared, the specific behavior and scope handling differ between formats.

Iterations in Recipe-Based Playbooks

In recipe-based playbooks, the loop retrieves the array to iterate from the iteration condition. Iterating on the array to support nested loops, the current element is appended to a dedicated stack structure named `iteration_element` inside the *global scope*. This stack-based approach ensures that each nesting level can independently access its own iteration variable without overwriting values from outer loops. Then The `execute` method is called for each step in the loop body. At the end of the internal loop the iteration element is popped from the stack.

Listing 6.4: Executing recipes loop extracted function

```
        for item in array_to_iterate:
            global_scope["iteration_element"].append(item)
            for el in self.steps:
                el.execute(executor,global_scope)
            global_scope["iteration_element"].pop()

```

CACAO-Compliant Custom `for` Loops

In CACAO compliant custom `for` loops, the iteration array is obtained by reading a variable and a specific field defined in the loop's configuration. Unlike recipe-based loops, iterating on the array the current element is stored directly in the *global scope* under a key specified in the `step_extensions`, eliminating the need for a stack. Then the `execute` method is called for each step in the loop body. This approach reduces complexity and simplifies variable management for nested loops.

Listing 6.5: Executing custom loop extracted function

```
        for element in array:
            global_scope[f"{self.condition['name_iteration_element']}"]=
    element
            for el in self.steps:
                el.execute(executor,global_scope)

```

CACAO-Compliant `while` Loops

The CACAO compliant `while` loop operates by repeatedly evaluating a condition defined in the loop configuration. As long as the condition evaluates to `true` in the current *global scope*, all steps in the loop body are executed in order. After each iteration, the condition is re-evaluated, allowing for dynamic exit based on changes in the scope during execution. This structure is especially suited for event-driven or state-dependent remediation procedures, where loop termination depends on achieving a particular system state.

Listing 6.6: Executing CACAO while loop extracted function

```
1         while self.condition.evaluate_condition(global_scope):
2             for el in self.steps:
3                 el.execute(executor, global_scope)
```

6.3.3 Conditional Execution: The `If` Step

The `If` step enables conditional branching within the execution flow. Its operation begins with the evaluation of a logical condition against the current `global_scope`, which contains the shared execution context.

If the condition evaluates to *true*, the step iterates sequentially over the set of actions defined in the *true branch*, invoking the `execute` method of each nested step. Conversely, if the condition evaluates to *false*, execution is redirected to the *false branch*, iterating over and executing its corresponding steps. The approach also allows nesting of conditional steps, enabling the representation of complex decision logic.

Listing 6.7: Executing If condition extracted function

```
1         if(self.condition.evaluate_condition(global_scope)):
2             for el in self.true_branch:
3                 el.execute(executor, global_scope)
4         else:
5             for el in self.false_branch:
6                 el.execute(executor, global_scope)
```

6.3.4 Concurrent Execution: The `Parallel` Step

The `Parallel` step facilitates concurrent execution of multiple independent branches. For each branch, a dedicated execution thread is instantiated using Python's `threading` library, with the branch logic encapsulated in a dedicated method (`execute_instance`).

The `execute` method of the `Parallel` step performs the following sequence:

1. Iterates over all configured branches (`next_steps`).
2. For each branch, initializes a new thread whose target is the `execute_instance` method, passing both the execution context and the branch's sequence of steps.

3. Starts all threads, enabling parallel execution of the branches.

4. Waits for all threads to complete (`join`), ensuring that the parallel step does not return control until every branch has finished execution.

Within `execute_instance`, the branch's steps are executed sequentially, each invoking its own `execute` method. This pattern allows multiple branches to progress in parallel, while preserving the sequential order of operations within each individual branch.

The parallel execution model is particularly suited for scenarios where independent tasks can be performed simultaneously, reducing total execution time and enabling better resource utilization.

Listing 6.8: Executing Parallel step extracted function

```
    def execute(self, executor, global_scope):
        threads = []
        for steps in self.next_steps:
            thread = Thread(target=self.execute_instance, args=(executor
    , global_scope, steps))
            thread.start()
            threads.append(thread)

        for t in threads:
            t.join()

    def execute_instance(self, executor, global_scope, steps):
        for el in steps:
            el.execute(executor, global_scope)
```

6.3.5 Graph-Based Execution Support

In addition to the standard `execute` method, the `If`, `Action`, and `Iteration_Loop` classes also support a graph-based execution mode through the `execute_graph` method.

For both `If` and `Iteration_Loop` steps, the control logic remains identical to that of the standard execution flow, with the only difference being that, for their internal steps, `execute_graph` is invoked instead of `execute`. This ensures that conditional and iterative constructs can be seamlessly integrated into a graph-based execution environment without altering their core decision or loop logic.

The `Action` step adopts a different strategy. In this case, there is no dedicated code identifier for the action type; instead, the resolution is performed dynamically based on the action's name, using a similar dictionary-based lookup mechanism. This design choice allows recipe authors to freely customize action names without requiring modifications to the underlying grammar definition (`grammar.tx`) avoiding additional parameters or rigid identifiers.

6.4 Condition Evaluation

Conditions play a central role in controlling execution flow within both `If` and `While` steps. As discussed in the previous chapter, the system supports three distinct types of conditions:

- **Recipe-defined conditions**:Boolean checks declared directly in the `recipe.rec` definition.
- **Custom conditions**:User-defined logical expressions with configurable operands and operators.
- **STIX pattern conditions**: CACAO compliant patterns evaluated using the `stix2matcher` library.

In the following, each category is described in detail.

6.4.1 Recipe-defined Conditions

These conditions perform a simple boolean check on a variable stored in the global execution scope. The evaluation process retrieves the variable value by name and compares its value against the logical negation flag (`value_not`) defined in the condition. This mechanism allows for expressions such as `if not "variable"`, where the `not` keyword reverses the boolean evaluation. The implementation return true if variable and value-not have two discord value, false otherwise By design, the logic covers all four combinations of variable value and negation flag.

```
Logic:
    var = True,  value_not = False  --> condition is True
    var = False, value_not = False  --> condition is False
    var = True,  value_not = True   --> condition is False
    var = False, value_not = True   --> condition is True
```

Listing 6.9: Recipe condition logic

```
1            return variable!=self.condition['value_not']
```

6.4.2 Custom Conditions

Custom conditions provide a flexible mechanism for evaluating arbitrary logical expressions defined within a recipe. A custom condition is composed of three main components:

- **Left-hand side (LHS)**: An expression whose value will be retrieved or computed from the current execution context.

- **Operator**: A symbolic comparator (e.g., `==`, `!=`, `>`) that maps to a specific Python function.
- **Right-hand side (RHS)**: A literal value or another variable reference to compare against the LHS.

The evaluation process follows three distinct steps:

1. **Parsing and evaluating the LHS expression**: The LHS is stored as a string. At runtime, it is evaluated in a restricted environment to prevent access to unsafe built-in functions or global variables.

Listing 6.10: Evaluating the left-hand side expression

```
lhs_value = eval(lhs_expr, {"__builtins__": {}}, global_scope)

```

If the expression references a variable not present in the global scope, the evaluation fails gracefully with a handled exception.

2. **Parsing the RHS value**: The RHS can represent either a hardcoded value or a variable reference. If the RHS is not a string, it is treated as a constant and returned unchanged. If it is a string, the parser checks whether it follows the special format var(<variable-name>). When this format is detected, the <variable-name> is extracted and looked up in the global scope dictionary. If the variable is not found, an explicit error is raised to prevent silent failures. Otherwise, the resolved value is returned. If the string does not match the var(...) syntax, it is treated as a hardcoded literal.

Listing 6.11: Resolving the right-hand side value

```
        if not isinstance(rhs, str):
            return rhs

        rhs = rhs.strip()
        if rhs.startswith("var(") and rhs.endswith(")"):
            var_name = rhs[4:-1].strip()
            return eval(var_name, {"__builtins__": {}}, scope)

```

3. **Applying the operator**: Operators are stored in a predefined dictionary mapping symbolic strings to Python callable functions from the `operator` module (or custom comparators if needed). This design ensures that all comparisons are performed in a controlled and consistent manner.

Listing 6.12: Operator mapping and comparison

```
OPERATORS = {
```

```
    "==": operator.eq,
    "!=": operator.ne,
    ">": operator.gt,
    "<": operator.lt,
    ">=": operator.ge,
    "<=": operator.le,
    "IN": lambda a, b: operator.contains(b, a)
}

result = OPERATORS[op_symbol](lhs_value, rhs_value)

```

This architecture allows the system to support a wide range of logical expressions without altering the recipe grammar. Furthermore, the separation between expression parsing, variable resolution, and operator application makes the implementation easy to extend.

Error handling plays a critical role: invalid expressions, missing variables, or unknown operators are caught and logged without interrupting the execution flow, preventing a single faulty condition from halting the recipe execution.

6.4.3 STIX Pattern Conditions

STIX pattern conditions allow CACAO compliant condition matching using structured cyber threat intelligence patterns. The evaluation process begins by extracting the STIX object name from the condition string. The corresponding object data is then retrieved from the execution data set and encapsulated into an `observed-data` STIX object. Using the `stix2matcher` library, the pattern is applied to determine whether the observed data matches the specified condition. This mechanism enables advanced, standards-based condition matching that integrates seamlessly with structured threat intelligence formats.

A STIX pattern condition is defined as a string containing a valid STIXpattern (e.g., `[file:hashes.'SHA-256' = 'abcd1234...']`), which is evaluated against runtime-collected data.

The evaluation process consists of four main steps:

1. **Extracting the STIX object name**: The first token of the pattern indicates the STIX object type (e.g., `file`, `ipv4-addr`). This information is parsed to identify which dataset entry should be retrieved from the execution context.

Listing 6.13: Extracting the STIX object name from the pattern

```
import re

match = re.match(r"\[(\w+):", pattern_str)
if not match:
    raise ValueError("Invalid STIX pattern")
object_name = match.group(1)

```

2. **Retrieving the object data**: The execution engine maintains a structured dataset of observed values. Once the object name is identified, the corresponding data is retrieved. If no data is found, the condition automatically evaluates to `False`.

Listing 6.14: Retrieving execution data for the STIX object

```
if object_name not in execution_data:
    return False
object_data = execution_data[object_name]

```

3. **Encapsulating data in an `observed-data` STIX object**: To comply with STIX specifications, the retrieved data is wrapped in an `observed-data` object before pattern evaluation. This ensures compatibility with libraries expecting fully structured STIX bundles.

Listing 6.15: Constructing a minimal observed-data object

```
observed_data = {
    "type": "observed-data",
    "id": f"observed-data--{uuid4()}",
    "objects": {
        "0": object_data
    }
}

```

4. **Evaluating the pattern with `stix2matcher`**: The `stix2matcher` library is used to evaluate the provided pattern against the constructed observed-data object. This library handles parsing, type checking, and logical evaluation of STIX patterns.

Listing 6.16: Evaluating the STIX pattern

```
from stix2matcher import match

try:
    match_result = match(pattern_str, observed_data)
except Exception as e:
    logger.error(f"STIX pattern evaluation failed: {e}")
    return False

```

This design enables complex, standards-based condition matching without requiring custom parsing logic for threat intelligence data. By relying on the STIX 2.1 specification and the `stix2matcher` library, the implementation remains aligned with widely adopted industry standards and is easily extensible to support new object types or pattern syntax as they are introduced.

Error handling ensures that malformed patterns, unsupported object types, or missing execution data do not halt the recipe execution, but instead produce a controlled `False` result with appropriate logging.

6.5 Generating a CACAO Playbook from an Executable Instance

The process of generating a CACAO compliant playbook from an executable instance is based on the orchestration of step definitions and their interconnections through unique identifiers (`UIID`). This mechanism ensures that the final playbook preserves both the logical flow and the conditional branching structure defined in the executable model.

The generation process starts with an initialization sequence:

Listing 6.17: Main generation loop for CACAO playbook steps

```
init_playbook_cacao(self, self.id_start)
add_start(self, self.id_start)
for el in self.steps:
    el.add_step_playbook_cacao(self.cacao, PB_uuid())
....
add_end(self, self.id_end)
...
```

The `add_step_playbook_cacao` method is invoked recursively for each step in the playbook. Since each step belongs to a specific class, the method implementation is delegated to the corresponding class definition.

6.5.1 UUID Assignment Mechanism

At the core of the generation process is a consistent UIID assignment strategy. A dedicated UIID handler is used to:

1. Assign the current UIID as the step's `id`.
2. Generate a new UIID for the `on_completion` (or `on_true`/`on_false`) field.
3. Pass this updated UIID context to subsequent steps.

This mechanism is executed during classes creation, to prepare identifiers for logging and error reporting during execution. For convenience, this process is also responsible for generating all the IDs needed to represent that type of step in a Cocoa playbook, such as on_completion, on_true, on_false, and next_step. This streamlines the subsequent process, as each step has all the information needed to generate its own step and, if necessary, recur.

For example, the `while` construct assigns its IDs as follows:

Listing 6.18: UUID assignment for a `while` loop

```
def add_cacao_IDs(self, pb_uuid):
    old_uuid = pb_uuid.get_uuid()
    pb_uuid.set_new_uuid()
    istance_id = {
        "id": old_uuid,
        "on_true": f"{pb_uuid.get_uuid()}",
    }
    for el in self.steps:
        el.add_cacao_IDs(pb_uuid)
    istance_id["on_completion"] = pb_uuid.get_uuid()
    ...
```

6.5.2 Step Generation for Control Structures

Once the IDs have been assigned, `add_step_playbook_cacao` can directly produce the step definition in CACAO format. For example, a `for` loop is represented as follows:

Listing 6.19: Generating CACAO representation for a `while` loop

```
def add_step_playbook_cacao(self,cacao,pb_uuid):
        cacao.append({
                "id": f"while-condition--{self.cacao_id['id']}",
                "type": "while-condition",
                "condition": self.condition,
                "on_true": f"{self.cacao_id['on_true']}",
                "on_completion": f"{self.cacao_id['on_completion']}",
            })
        for el in self.steps:
            el.add_step_playbook_cacao(cacao,pb_uuid)
            ...
```

This recursive approach ensures that:

1. All nested steps are correctly expanded into CACAO format.
2. Execution flow links (`on_true`, `on_completion`, etc.) are preserved.
3. Conditional and iterative constructs retain their semantics in the final playbook.

6.5.3 Special Handling of End-Steps in Conditional and Parallel Instructions

A particular consideration is required for the `end-step` logic in `if` and `parallel` instructions. In these cases, the notion of "end" does not simply mark the termination of the overall playbook execution, but rather the end of a specific branch of execution. For `if` statements, this refers to the end of the `true-false` branch, while in `parallel` steps it refers to the completion of all concurrently executed branches. This distinction must be explicitly addressed at the time of generating the CACAO playbook.

In practice, each `add_step_playbook_cacao` method concludes by executing an instruction of the form:

```
pb_uuid.set_uuid(self.cacao_id["on_completion"])
```

This operation sets the current UIID to the value of the `on_completion` field, which had already been assigned during the initial ID generation phase. As a consequence, for `if` and `parallel` classes, after the recursive processing of nested steps, the `on_completion` value from the last executed step becomes the UIID to be assigned to the corresponding `end-step`.

At first glance, one might expect that overwriting the UIID in this way would compromise the consistency of subsequent steps. However, it is important to note that each class already retains all the necessary ID information for its own steps. Consequently, they no longer depend on the `pb_uuid` instance for further ID generation, except for the specific purpose of assigning the correct UIID to the `end-step` as described above.

Listing 6.20: Generating CACAO parallel step

```
def add_step_playbook_cacao(self, cacao, pb_uuid):

        id_on_completion = self.cacao_id["on_completion"]

        # Add the main parallel step to the playbook
        cacao.append({
            "id": f"parallel--{self.cacao_id['id']}",
            "name": self.name,
            "type": "parallel",
            "next_steps": self.cacao_id["next_steps"],
            "on_completion": id_on_completion
        })

        #iterate over each branch in next_steps
        for branch_index, branch in enumerate(self.next_steps, start=1):
            for step in branch:
                step.add_step_playbook_cacao(cacao, pb_uuid)

            old_uuid = pb_uuid.get_uuid()
            pb_uuid.set_new_uuid()

            # Add an 'end' step for each branch
            cacao.append({
                "id": f"end--{old_uuid}",
                "type": "end",
                "name": f"end of branch number {branch_index} for
    parallel_step:{self.cacao_id['id']}"
            })

        # Restore the UUID generator to point to the on_completion step
        pb_uuid.set_uuid(id_on_completion)
```

Chapter 7

Validation and Testing

This chapter presents the validation and testing activities carried out in two distinct environments: *on-premises* and *cloud*.

For the *on-premises* setup, as previously discussed, it is inherited from the previous tool set of `recipe.rec` files were used to perform various actions. These recipes will be accompanied by the corresponding execution results, which include modifications to the graph, generated `iptables` rules, or, in some cases, simple log messages.

In the *cloud* environment, validation relied on custom CACAO playbooks designed to resemble realistic operational scenarios. The outcomes in this case include network policies enforced with the `Cilium` CNI, modifications to the cluster nodes or pods, KubeArmor policies, as well as log messages.

7.1 On-premises Validation

The validation in the *on-premises* environment was performed through a set of (inherited from RR-tool) `recipe.rec` files, each designed to emulate different security scenarios. These recipes range from simple actions, such as adding a filtering rule to a firewall, to more advanced operations like deploying a honeypot. Furthermore, they also include more complex scenarios, where additional firewalls are introduced (when necessary) and specific filtering rules are enforced along all possible paths between an attacker and a victim.

In the following, are some representative recipes with their effects.

7.1.1 Filtering a Malicious IP

The following recipe demonstrates how a malicious IP can be filtered after ensuring that all existing paths between a victim and an attacker are properly protected by firewalls. If a firewall with `level_4_filtering` capabilities is already present along a path, a filtering rule is simply added to it. Otherwise, a new firewall is inserted behind the impacted host, and the corresponding filtering rule is applied to the newly created node.

```
list_paths from impacted_host_ip to attacker
iterate_on path_list
    find_node of type 'firewall' in iteration_element
    with 'level_4_filtering'
    if found
        add_filtering_rule rule_level_4 to found_node
    else
        add_firewall behind impacted_host_ip in iteration_element
        with 'level_4_filtering'
        add_filtering_rule rule_level_4 to new_node
    endif
end_iteration
```

The graphs before and after the execution of the recipe are shown (in case the remediation is applied on a graph with `interpret_graph`) in figures 7.1 - 7.2, together with the generated `iptables` rules, produced by providing the malicious IP as input, and the relevant log messages displayed during the execution in Figure 7.3.

Similar considerations are made for the recipe called monitor-traffic.rec.

```
list_paths from impacted_host_ip to attacker
iterate_on path_list
find_node of type 'network_monitor' in iteration_element
if not found
add_network_monitor behind impacted_host_ip in iteration_element
endif
end_iteration
```

In this case, a monitor-traffic node is added between the victim and the attacker. In figure 7.4 the graph obtained after execution.

7.1.2 Deploying a Honeypot

The following recipe illustrates the deployment of a honeypot to the network. In this case, the recipe iterates over the impacted nodes and attaches a honeypot configured with an `apache_vulnerability`. This action is executed exclusively on the graph: it does not generate filtering rules or logs, but instead results in structural changes to the network topology representation.

```
iterate_on impacted_nodes
    add_honeypot with 'apache_vulnerability'
end_iteration
```

The graph in the figure 7.5 illustrates the network after the recipe's execution, highlighting the introduction of the honeypot node. The initial configuration can be found in the image 7.1.

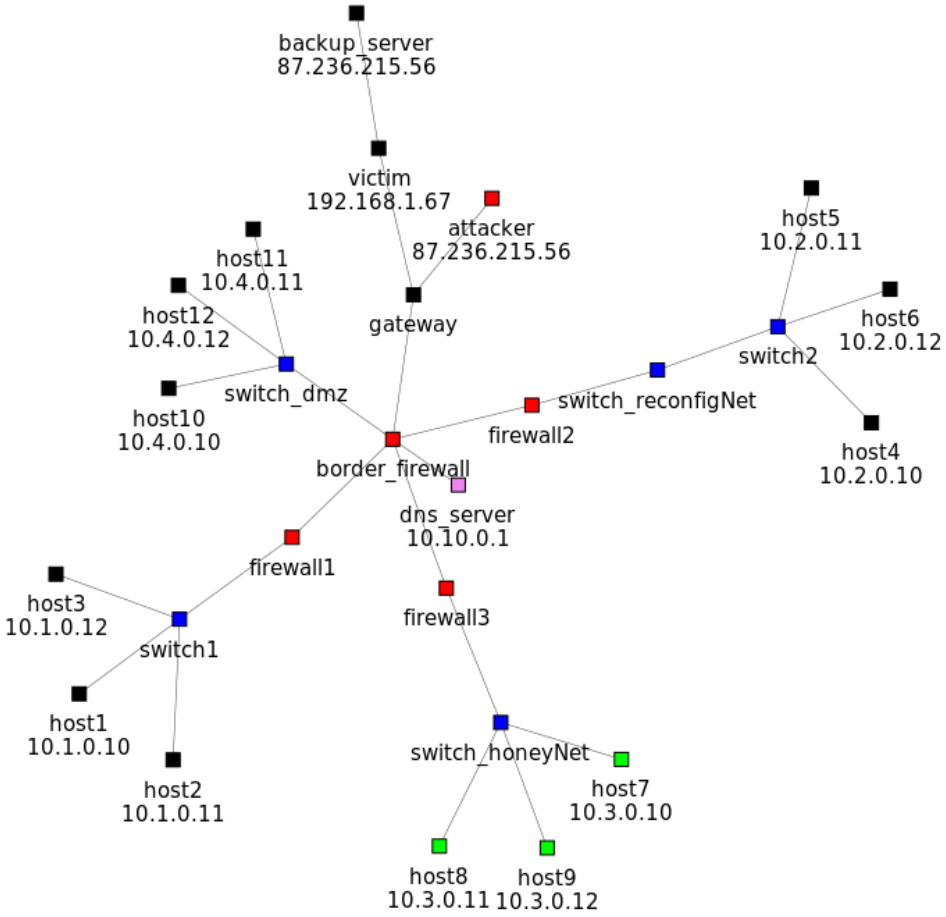

Figure 7.1: Initial configuration

7.1.3 Moving Nodes into a Reconfiguration Network

The following recipe, named `put_into_reconfiguration.rec`, performs a structural modification on the graph by moving the impacted nodes into a dedicated network called `reconfiguration_net`. This operation is intended to represent a reconfiguration step, where compromised or impacted nodes are logically isolated from the main topology and relocated into a separate environment for further analysis or remediation.

```
iterate_on impacted_nodes
    move iteration_element to 'reconfiguration_net'
```

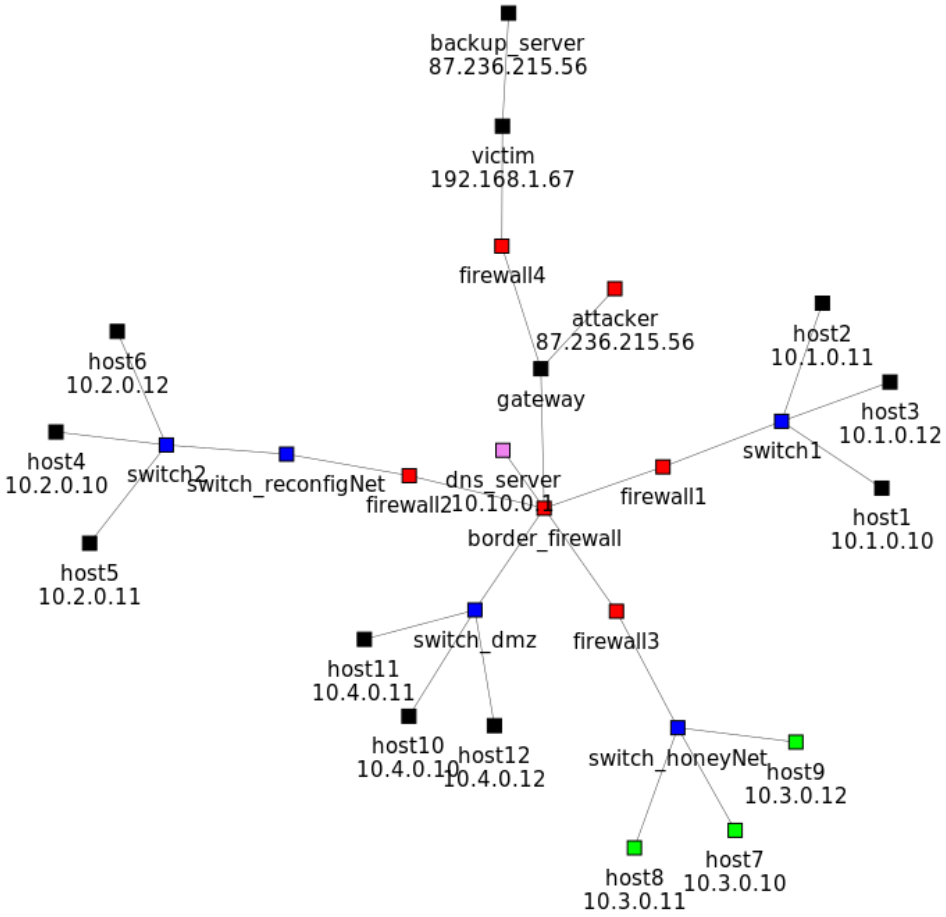

Figure 7.2: Firewall added between victim and attacker

```
end_iteration
```

As this action is executed solely on the graph, the effects are visible as structural changes in the topology. In the figure 7.6, the graph is shown after the execution of the recipe, clearly illustrating the relocation of the "victim" node into the `reconfiguration_net`. The initial configuration can be found in the image 7.1.

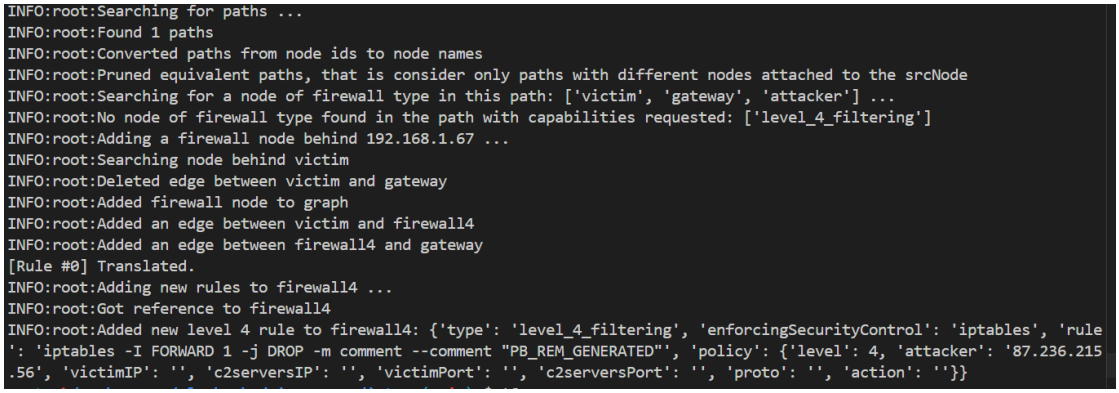

Figure 7.3: filter_ip_port.rec logs

7.2 Cloud Environment Testing

To validate the tool in a cloud environment, a series of CACAO playbooks was developed to test all the implemented functionalities. Among these, some representative playbooks are presented in detail, together with a few examples of `KubeArmor` policies automatically generated from the `.rec` files.

The results of their execution are reported, including relevant log messages and visual representations of the cluster state. This approach highlights how the tool behaves not only under controlled testing conditions but also when applied to settings that resemble real-world environments.

7.2.1 Quarantine pod

The following example illustrates a CACAO playbook designed for cloud-native environments, with a focus on containerized workloads. Its objective is to quarantine potentially compromised application pods while limiting their ability to spread malicious activity across the cluster.

At a high level, the playbook executes the following workflow:
Input:Detected-suspicious-pod.

1. Retrieve all pods associated with the affected label.
2. If at least one pod is found:
   (a) Block communication with known malicious IP addresses for the detected-pod label matching pod.
   (b) Deny egress traffic by applying a network policy that restricts outbound communication for pods with the label app: quarantined-frontend.
   (c) For each affected pod:

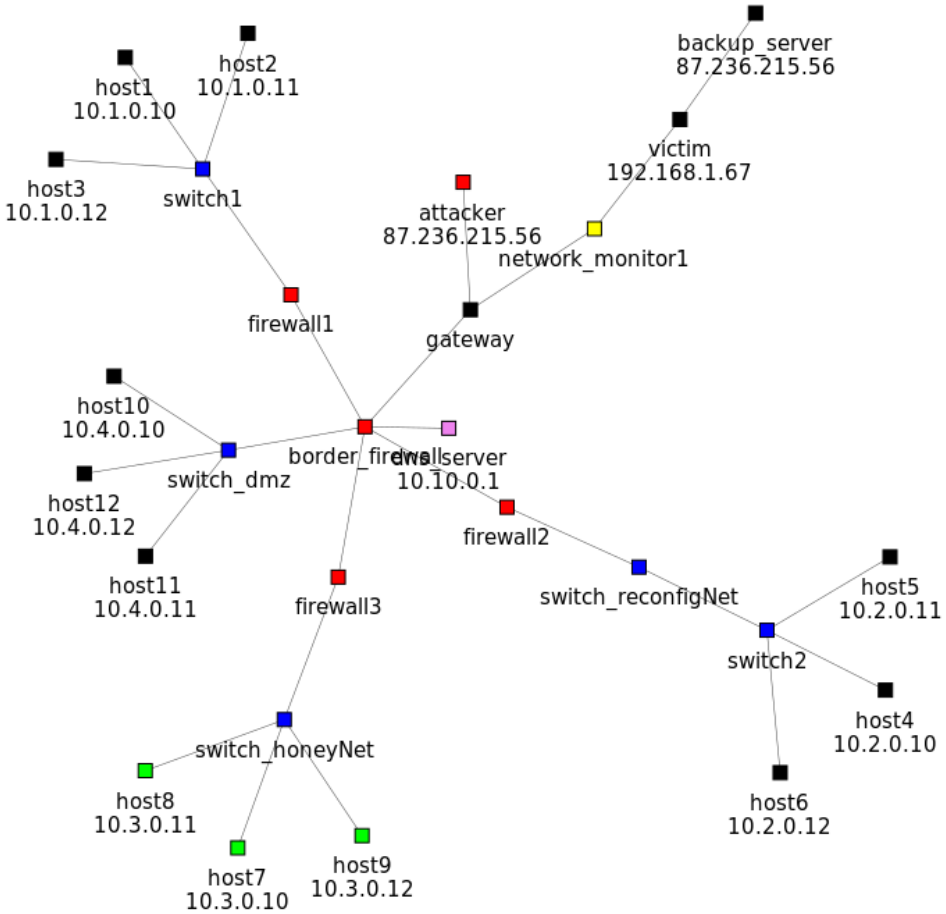

Figure 7.4: Added monitor-traffic

i. Update the pod's labels (e.g., reassigning the app label to *quarantined-frontend*) to reflect its quarantined state also in the system graph.

ii. Add tolerations for the label *quarantined_frontend* to allow proper scheduling of quarantined workloads.

(d) Apply a `NoExecute` taint on the nodes using a specified key, ensuring that only pods with the matching toleration can remain scheduled on those nodes.

3. If no affected pods are found, generate an alert log entry.

This playbook exemplifies how structured remediation logic can be automated in a

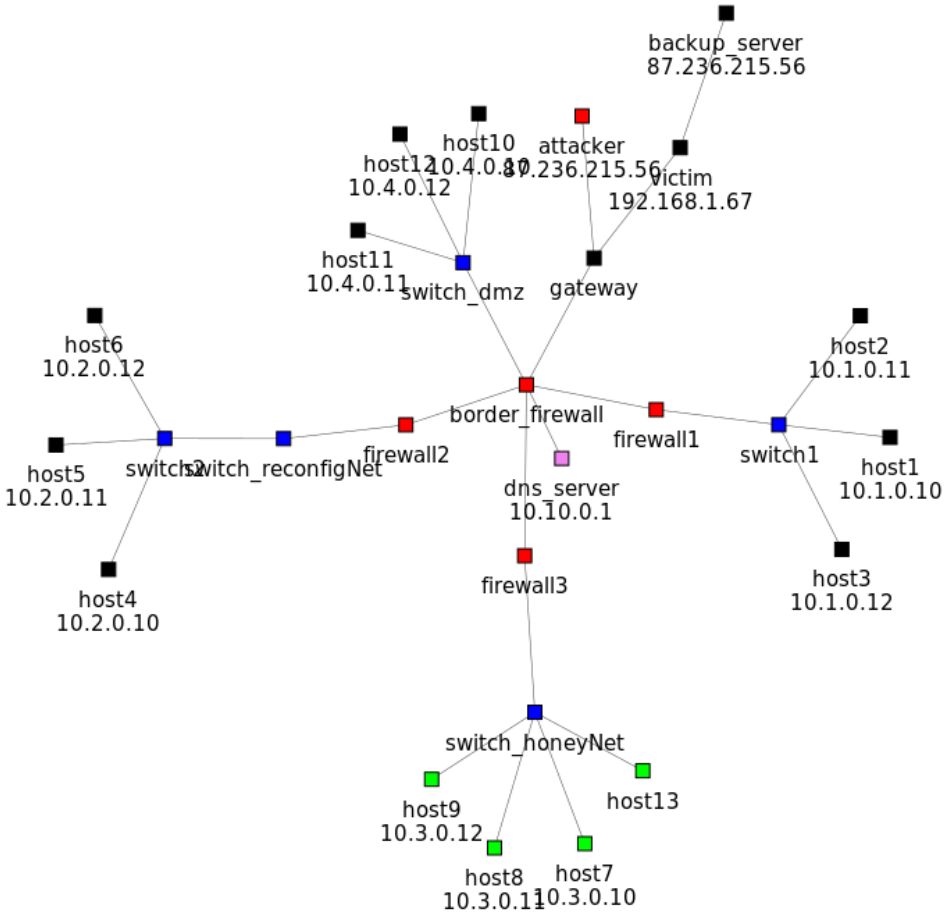

Figure 7.5: Adding honeypot

Kubernetes environment. By combining conditional execution with isolation mechanisms, such as label updates, network restrictions, and node taints, it provides a systematic method for containing compromised workloads.

Such an approach is particularly useful in scenarios where:

- Rapid containment of a suspected compromise is necessary to prevent lateral movement within the cluster.
- Security teams need to enforce automated quarantine procedures without requiring manual intervention.

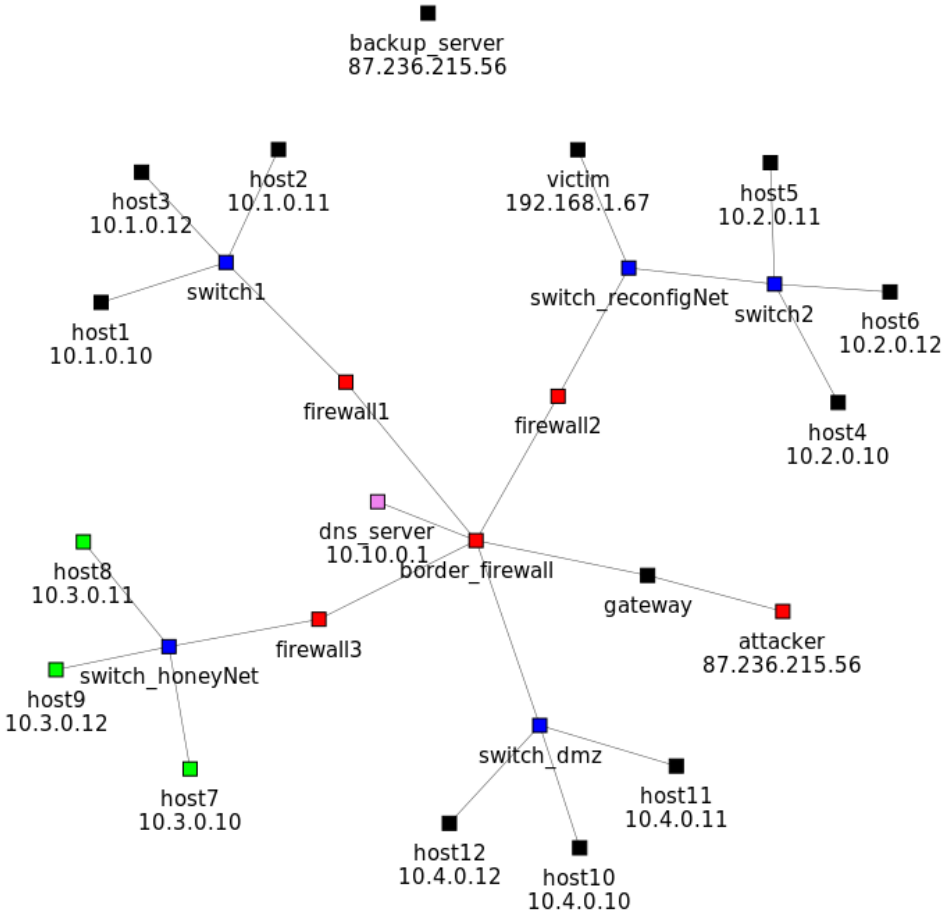

Figure 7.6: moving impacted node

- Cloud-native infrastructures with high workload turnover demand consistent and repeatable remediation actions.

In the figure 7.7 the policies applied to the cluster are shown while in the figure 7.8 it can be noted how the only pod in running state is the detected pod (in its old version), the only one in fact with the label updated to `quarantined_frontend`

```
C:\Users\SimoD>kubectl get ciliumnetworkpolicies -A
NAMESPACE   NAME                 AGE
default     block-all-traffic    18d
default     deny-egress-to-ip    8d
```

Figure 7.7: Applied policies

```
C:\Users\SimoD>kubectl get pods
NAME                                 READY   STATUS    RESTARTS        AGE
cartservice-696db49464-l22f9         0/1     Pending   0               104s
checkoutservice-7698c9f5dd-4qqwd     0/1     Pending   0               104s
currencyservice-847cdd6564-tbslq     0/1     Pending   0               104s
emailservice-6d88759bcf-czm5z        0/1     Pending   0               103s
frontend-79c5d69444-9qvdr            0/1     Pending   0               103s
frontend-79c5d69444-w99qs            1/1     Running   1 (7m47s ago)   4d6h
loadgenerator-69b76b8db7-g45rc       0/1     Pending   0               104s
paymentservice-6f495db55f-mjkgg      0/1     Pending   0               104s
productcatalogservice-75644cc8f5-ck224   0/1     Pending   0               104s
recommendationservice-5f9fbb6c4d-8rdgz   0/1     Pending   0               104s
redis-cart-647b66f47f-njvkc          0/1     Pending   0               104s
shippingservice-6b9848c59f-xg57h     0/1     Pending   0               104s

C:\Users\SimoD>kubectl get pod frontend-79c5d69444-w99qs --show-labels
NAME                        READY   STATUS    RESTARTS        AGE    LABELS
frontend-79c5d69444-w99qs   1/1     Running   1 (7m53s ago)   4d6h   app=quarantined_frontend,pod-template-hash=79c5d694
44,skaffold.dev/run-id=6ca48125-1e3b-450c-a1df-2faa89c70608
```

Figure 7.8: Cluster setup after quarantine action is executed (only the quarantined pod is in running state))

7.2.2 Deployment-level Remediation Playbook

This CACAO playbook targets the remediation of suspicious workloads at the deployment level in a Kubernetes cluster. Its main goal is to promptly evict potentially compromised pods and remove the higher-level deployment object that manages them, thus preventing automatic re-creation of malicious or compromised containers.

At a high level, the workflow is defined as follows:

1. Extract the label from the suspicious pod and retrieve all pods associated with that label.
2. For each affected pod:
   (a) Retrieve and log pod metadata for auditing and forensic purposes.
   (b) Evict the pod from the cluster.
3. Delete the corresponding deployment to ensure that no further replicas of the compromised application are scheduled.

This type of playbook is particularly useful in scenarios where a single workload has been identified as compromised and is under the control of a higher-level deployment object. Simply removing pods would not be sufficient, since Kubernetes automatically recreates them to maintain the desired deployment state. By explicitly deleting the deployment after evicting its pods, this playbook ensures that the entire application instance is dismantled and cannot continue operating in a potentially harmful way.

A realistic use case for this remediation strategy is the detection of a container running a malicious image pulled from an untrusted registry.

In this case, the detected suspicious pod belongs to the `frontend` application, while the deployment scheduled for deletion corresponds to the `payment-service`.

The metadata retrieved from the suspicious pod is the following:


```
{
  'name': 'frontend-79c5d69444-9qvdr',
  'namespace': 'default',
  'labels': {
      'app': 'frontend',
      'pod-template-hash': '79c5d69444',
      'skaffold.dev/run-id': '6ca48125-1e3b-450c-a1df-2faa89c70608'
  },
  'annotations': {
      'sidecar.istio.io/rewriteAppHTTPProbers': 'true'
  },
  'owner_references': [
      {
        'kind': 'ReplicaSet',
        'name': 'frontend-79c5d69444',
        'uid': 'dd11ea7e-592f-47cc-9379-1c58daa82e07'
      }
  ]
}
```


Figure 7.9 shows that the `payment-service` pod has been fully deleted following the deployment removal, while the `frontend` pod was correctly evicted and then automatically recreated by the Google Application daemon. This behavior highlights how the eviction only affects the individual pod instance, while the higher-level deployment controller ensures that a replacement pod is immediately scheduled, resulting in a new pod identifier.

7.2.3 Node-level Remediation Playbook

This CACAO playbook targets remediation actions at the Kubernetes node level, with the goal of temporarily isolating a node, assessing its workloads, and applying network restrictions to pods that are deemed untrusted. The strategy is designed to limit potential lateral movement and data exfiltration while keeping the node under controlled conditions.

At a high level, the workflow proceeds as follows: Input: a detected suspicious pod and metadata about the node hosting it.

1. Apply a `cordon` operation on the node, preventing the scheduling of new pods onto it.

```
C:\Users\SimoD>kubectl get pods
NAME                                    READY   STATUS     RESTARTS   AGE
cartservice-696db49464-l22f9            0/1     Running    0          16h
checkoutservice-7698c9f5dd-4qqwd        1/1     Running    0          16h
currencyservice-847cdd6564-tbslq        0/1     Running    0          16h
emailservice-6d88759bcf-czm5z           0/1     Running    0          16h
frontend-79c5d69444-9qvdr               1/1     Running    0          16h
loadgenerator-69b76b8db7-g45rc          0/1     Init:0/1   0          16h
paymentservice-6f495db55f-mjkgg         1/1     Running    0          16h
productcatalogservice-75644cc8f5-ck224  1/1     Running    0          16h
recommendationservice-5f9fbb6c4d-8rdgz  0/1     Running    0          16h
redis-cart-647b66f47f-njvkc             1/1     Running    0          16h
shippingservice-6b9848c59f-xg57h        1/1     Running    0          16h

C:\Users\SimoD>kubectl get pods
NAME                                    READY   STATUS    RESTARTS        AGE
cartservice-696db49464-l22f9            1/1     Running   0               17h
checkoutservice-7698c9f5dd-4qqwd        1/1     Running   0               17h
currencyservice-847cdd6564-tbslq        1/1     Running   0               17h
emailservice-6d88759bcf-czm5z           1/1     Running   1 (4m58s ago)   17h
frontend-79c5d69444-gzwld               1/1     Running   0               39s
loadgenerator-69b76b8db7-g45rc          1/1     Running   0               17h
productcatalogservice-75644cc8f5-ck224  1/1     Running   0               17h
recommendationservice-5f9fbb6c4d-8rdgz  1/1     Running   1 (4m58s ago)   17h
redis-cart-647b66f47f-njvkc             1/1     Running   0               17h
shippingservice-6b9848c59f-xg57h        1/1     Running   0               17h
```

Figure 7.9: deplyoment deletion

2. Retrieve all pods sharing the label of the detected pod.
3. For each pod retrieved:
   (a) Extract the pod specification.
   (b) If the pod is marked as affected (i.e., the attribute `trusted` is not set to `True`), apply network restrictions by denying egress traffic.
4. Finally, apply an `uncordon` operation to the node, restoring its scheduling capability.

This remediation strategy is useful in scenarios where a specific node is suspected of hosting compromised workloads, yet a full node shutdown or drain would be too disruptive. By cordoning the node first, the cluster prevents additional workloads from being scheduled while security checks are carried out. The playbook then selectively applies containment measures (e.g., blocking egress) to pods that are not explicitly trusted, thereby reducing the attack surface without immediately disrupting benign workloads. To demonstrate the effectiveness of the node-level remediation playbook, we provide an execution example.

In the figure 7.9, the node hosting the suspicious workload is successfully marked as `cordoned`, ensuring that no new pods can be scheduled onto it during the inspection phase. Subsequently, we retrieve the specification of the suspicious pod. The following metadata corresponds to the `frontend` pod:

```
C:\Users\SimoD>kubectl get nodes
NAME       STATUS                     ROLES           AGE   VERSION
minikube   Ready,SchedulingDisabled   control-plane   19d   v1.32.0
```

Figure 7.10: cordoned minikube node

```
{
  'name': 'frontend-79c5d69444-gzwld',
  'namespace': 'default',
  'containers': {
      'name': 'server',
      'image': 'frontend:cf3336a24884053da930cad7610b25fbc3516a867d7f9a
      026e105e2b4ce1f06b',
      'resources': {
          'limits': {
              'cpu': '200m',
              'memory': '128Mi'
          },
          'requests': {
              'cpu': '100m',
              'memory': '64Mi'
          }
      },
      'status': 'approved'
  }
}
```

In this environment, only the following test container images are considered approved:

```
approved_images = [
    "gcr.io/my-project/frontend:latest",
    "gcr.io/my-project/frontend:v1.2.3"
]
```

Since the running image does not match any entry in the approved list, the pod is flagged as untrusted (i.e., the `trusted` parameter is set to `False`). As a result, a *deny-all* network policy is applied to block all egress traffic from the pod.

Finally, a verification step is shown: by attaching an ephemeral pod to the `frontend` workload and attempting to ping external IP addresses such as `1.1.1.1` and `8.8.8.8`, it is demonstrated that outbound traffic is indeed blocked (Figure 7.11). This confirms that the remediation logic successfully enforces containment on unapproved workloads, while the node itself can later be safely uncordoned to resume normal cluster operations.

```
C:\Users\SimoD>kubectl debug -it frontend-79c5d69444-gzwld --image=nicolaka/netshoot --target=server
Targeting container "server". If you don't see processes from this container it may be because the container runtime doesn't support this feature.
Defaulting debug container name to debugger-krdwg.
If you don't see a command prompt, try pressing enter.
frontend-79c5d69444-gzwld% ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
^C
--- 1.1.1.1 ping statistics ---
14 packets transmitted, 0 received, 100% packet loss, time 13308ms

frontend-79c5d69444-gzwld% ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
^C
--- 8.8.8.8 ping statistics ---
10 packets transmitted, 0 received, 100% packet loss, time 9205ms

frontend-79c5d69444-gzwld%
```

Figure 7.11: applied deny all policy

7.3 KubeArmor Policy Support

As previously mentioned, the tool also provides limited support for the generation of KubeArmor policies. In this section, a few representative policies are presented as examples.

It is important to note that, unlike other remediation mechanisms, the tool does not support the direct enforcement of KubeArmor policies. Instead, the actual enforcement layer has been delegated to Cilium, which is better suited for managing and enforcing network-level rules.

Due to the intrinsic nature and scope of KubeArmor, not all remediation actions described in the recipe-based approach can be implemented. Nevertheless, in order to maintain conceptual continuity and to preserve opportunities for future development, an effort has been made to express policies that resemble, as closely as possible, the SDN-like behaviors described earlier.

7.3.1 Isolation Policy with KubeArmor

When the remediation file (`.rec`) contains an action of type `isolate` and the selected agent is `container-security`, the tool generates a corresponding KubeArmor policy. This policy is instantiated from a predefined template, which is filled at runtime with the contextual parameters of the execution environment, such as the namespace and the pod label.

The generated policy is shown below:

```
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: deny-all-network
  namespace: {{NAMESPACE}}
spec:
  selector:
    matchLabels:
      app: {{POD_LABEL}}
  network:
    matchProtocols:
```

```
      - protocol: TCP
      - protocol: UDP
  action: Block
```

This policy enforces a complete network isolation of the selected pod. Specifically, it denies both Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) traffic, regardless of the direction or target, effectively isolating the pod from the rest of the cluster and the external network.

Unlike the `iptables`-based approach used in other agents, KubeArmor policies do not operate at the raw IP layer. Therefore, it is not possible to express fine-grained rules such as `DROP` for specific IP addresses. Instead, the isolation is implemented at the process and protocol level, providing a coarse-grained but effective mechanism to contain potentially compromised workloads.

7.3.2 Network Monitoring Policy with KubeArmor

When the remediation file (`.rec`) contains an action of type `add_network_monitor` and the selected agent is `container-security`, the tool generates a dedicated KubeArmor policy. As in the previous case, this policy is instantiated from a template that is dynamically filled at runtime with contextual parameters such as the namespace, pod label, and network path.

The generated policy is the following:

```
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: network-monitor-policy
  namespace: {{ NAMESPACE }}
spec:
  severity: 5
  message: "Monitoring TCP network activity on {{IMPACTED_NODE}}"
  selector:
    matchLabels:
      app: {{POD_LABEL}}
  network:
    matchProtocols:
      - protocol: TCP
        fromSource:
          - path: {{NETWORK_PATH}}
  action: Audit
```

This policy does not block or restrict network traffic, but instead puts the specified pod under monitoring. More precisely, it traces TCP connections initiated from the executable defined in `NETWORK_PATH`, and logs these events in audit mode. This allows the operator to observe network activity patterns without actively enforcing restrictions.

7.3.3 Translation of Traffic Control Actions

Some actions present in the remediation recipes, such as *allow_traffic* or *add_filtering_rule*, cannot be directly implemented in KubeArmor due to its process-centric design. Instead, these actions are translated into policies that either allow or block network traffic originating from specific executables.

For example, the following KubeArmor network specification: permits or denies TCP traffic depending on the policy's `action` field. In this way, process-level enforcement simulates the effect of filtering traffic between nodes or applying firewall rules in the SDN/recipe model, while remaining compatible with KubeArmor's capabilities.

```
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: Allow TCP traffic from curl and python3
  namespace: {{ NAMESPACE }}
spec:
  selector:
    matchLabels:
      app: {{ POD_LABEL}}
network:
  matchProtocols:
    - protocol: TCP
      fromSource:
        - path: /bin/curl
        - path: /usr/bin/python3
action: Allow
```

Chapter 8

Conclusion and Future Work

This thesis presented the design and development of a tool for automated incident response, capable of translating alerts into executable actions. The main contributions of the work were twofold: first, the definition of executable programmatic instances of playbooks compliant with the CACAO standard, and second, the implementation of remediation actions specifically tailored for kubernetes-based environments. The tool was validated in realistic scenarios, demonstrating that it can effectively bridge the gap between the abstract representation of response procedures and their practical execution on both cloud-native and on-premises infrastructures.

From a practical perspective, the tool enables security teams to automate common remediation tasks, such as node management, traffic filtering, and container-level operations, through structured playbooks. The validation confirmed its ability to execute complete workflows correctly, ensuring that remediation can be applied consistently and reproducibly. By abstracting playbook definitions into executable instances, the tool supports interoperability and fosters the adoption of standardized approaches to incident response.

Nevertheless, some limitations emerged during the development. The process of mapping from alerts to appropriate corrective actions is based on static information. In addition, support for Kubearmor security policies is currently partial, limiting the range of possible remediation strategies. Addressing these aspects would further strengthen the robustness and adaptability of the proposed solution.

Future work will focus primarily on refining the alert-to-remediation mapping process, potentially incorporating artificial intelligence techniques to improve accuracy and adaptability. Another important direction is the extension of supported actions, both through native python APIs and the integration of additional enforcement mechanisms. In particular, extending the current capabilities with advanced Cilium policies and supporting runtime protection frameworks such as Kubearmor will broaden the scope of applicable use cases.

Bibliography

[1] Henrik Karlzen and Teodor Sommestad. «Automatic incident response solutions: a review of proposed solutions' input and output». In: *Proceedings of the 18th International Conference on Availability, Reliability and Security*. ARES '23. Benevento, Italy: Association for Computing Machinery, 2023. ISBN: 9798400707728. DOI: 10.1145/3600160.3605066. URL: https://doi.org/10.1145/3600160.3605066.

[2] Y et al. «PHOENI2X – A European Cyber Resilience Framework...» In: (2023).

[3] European Union Agency for Cybersecurity (ENISA). *Good Practice Guide for Incident Management*. Tech. rep. ENISA, 2011. URL: https://www.enisa.europa.eu/sites/default/files/publications/Incident_Management_guide.pdf.

[4] Content Team, Axur. *The Evolution of the SOC: From Traditional SOC to AI Powered SOC 3.0*. Blog post. https://blog.axur.com/en-us/evolution-of-soc-ai-driven-soc-3-0 (accessed on October 15, 2025). Mar. 2025.

[5] Martin Husák and Milan Čermák. «SoK: Applications and Challenges of using Recommender Systems in Cybersecurity Incident Handling and Response». In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ARES '22. Vienna, Austria: Association for Computing Machinery, 2022. ISBN: 9781450396707. DOI: 10.1145/3538969.3538981. URL: https://doi.org/10.1145/3538969.3538981.

[6] Keith Stouffer, Joe Falco, and Karen Scarfone. *Guide to Industrial Control Systems (ICS) Security (Draft NIST SP 800-82 Rev. 2)*. Tech. rep. http://www.gocs.com.de/pages/fachberichte/archiv/164-sp800_82_r2_draft.pdf. National Institute of Standards and Technology (NIST), 2014.

[7] Patrick Empl, Horst Reiser, and Konstantin Böttinger. «Standard-based Automation of Vulnerability Remediation in ICS Using CACAO and CSAF». In: *Proceedings of the 18th International Conference on Availability, Reliability and Security (ARES)* (2023).

[8] IBM. *Cos'è SOAR (Security Orchestration, Automation and Response)?* https://www.ibm.com/it-it/topics/security-orchestration-automation-response. Accessed: 2025-08-07. 2023.

[9] David Roche and Seamus Dowling. «Elevating Cybersecurity Posture by Implementing SOAR». In: *2023 Cyber Research Conference - Ireland (Cyber-RCI)*. 2023, pp. 1–7. DOI: 10.1109/Cyber-RCI59474.2023.10671437.

[10] Daniel Schlette, Marco Caselli, and Günther Pernul. «A Comparative Study on Cyber Threat Intelligence: The Security Incident Response Perspective». In: *IEEE Communications Surveys and Tutorials* PP (Oct. 2021), pp. 1–1. DOI: 10.1109/COMST.2021.3117338.

[11] OASIS Collaborative Automated Course of Action Operations (CACAO) TC. *CACAO Security Playbooks Version 2.0*. Tech. rep. cs01. Committee Specification 01. OASIS Open, May 2024. URL: https://docs.oasis-open.org/cacao/security-playbooks/v2.0/cs01/security-playbooks-v2.0-cs01.pdf.

[12] OASIS. *STIX Version 2.1. Part 5: STIX Patterning*. https://docs.oasis-open.org/cti/stix-pattern/v2.1/stix-pattern-v2.1.html. Accessed: 2025-08-01. 2021.

[13] MITRE Corporation. *MITRE ATT&CK Framework*. https://attack.mitre.org/. Accessed August 2025. 2024.

[14] Francesco Settanni, Leonardo Regano, Cataldo Basile, and Antonio Lioy. «A Model for Automated Cybersecurity Threat Remediation and Sharing». In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. 2023, pp. 492–497. DOI: 10.1109/NetSoft57336.2023.10175486.

[15] Francesco Settanni. «Towards intelligence driven automated incident response». PhD thesis. Politecnico di Torino, 2022.

[16] Cataldo Basile, Daniele Canavese, Leonardo Regano, Ignazio Pedone, and Antonio Lioy. «A model of capabilities of Network Security Functions». In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 2022, pp. 474–479. DOI: 10.1109/NetSoft54395.2022.9844057.

[17] TNO and COSSAS Project. *SOARCA - Security Orchestrator for Advanced Response to Cyber Attacks*. https://github.com/COSSAS/SOARCA. Accessed: 2025-08-01. 2024.

[18] Open Cybersecurity Alliance. *CACAO Roaster - Web Interface for CACAO v2.0 Playbooks*. https://github.com/opencybersecurityalliance/cacao-roaster. Accessed: 2025-08-01. 2024.

[19] Vasileios Mavroeidis and Mateusz Zych. *Cybersecurity Playbook Sharing with STIX 2.1*. 2022. arXiv: 2203.04136 [cs.CR]. URL: https://arxiv.org/abs/2203.04136.

[20] Google Cloud Platform. *Online Boutique: Cloud-Native Microservices Demo Application*. Accessed: 2025-08-09. 2025. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

Acronyms

ABAC
Attribute-Based Access Control

AI
artificial intelligence

API
Application Programming Interface

APT
Advanced Persistent Threat

CACAO
Collaborative Automated Course of Action Operations

CERT
Computer Emergency Response Team

CNI
Container Network Interface

COPS
Collaborative Open Playbook Standard

CSAF
Common Security Advisory Framework

CSIRT
Computer Security Incident Response Team

CTI
Cyber Threat Intelligence

DCS
Distributed Control System

DSL
domain-specific language

eBPF
extended Berkeley Packet Filter

EDR
Endpoint Detection and Response

ENISA
European Union Agency for Cybersecurity

GKE
Google Kubernetes Engine

HMI
Human Machine Interface

HTTP
HyperText Transfer Protocol

IACD
Integrated Adaptive Cyber Defense

ICS
Industrial Control System

IDS
Intrusion Detection System

IoC
Indicator of Compromise

IoT
Internet of Things

IP
Internet Protocol

IPS
Intrusion Prevention System

iptables
IP Tables Firewall

IR
Incident Response

IT
Information Technology

JAR
Java ARchive

JSON
JavaScript Object Notation

MAC
Media Access Control

MANO
Management and Orchestration

MITRE ATT&CK
MITRE Adversarial Tactics, Techniques, and Common Knowledge

MITRE DEFEND
MITRE Defensive Engagement Framework

MQTT
Message Queuing Telemetry Transport

NDR
Network Detection and Response

NSF
Network Security Function

OASIS
Organization for the Advancement of Structured Information Standards

OPENC2
Open Command and Control

OT
Operational Technology

PLC

Programmable Logic Controller

RE&CT

Requirements for the Expression of Cybersecurity Techniques

REC

Recovery file format

RECAST

Resilient Event Conditions Action System against Threats

RR

Recommendation and Remediation

SCADA

Supervisory Control and Data Acquisition

SDN

Software-Defined Networking

SIEM

Security Information and Event Management

SoK

Systematization of Knowledge

SOAR

Security Orchestration, Automation, and Response

SOARCA

Security Orchestrator for Advanced Response to Cyber Attacks

SOC

Security Operations Centre

SPL

STIX Patterning Language

SSH

Secure Shell

STIX

Structured Threat Information Expression

SUPC
SDN Unified Policy Configuration

TAXII
Trusted Automated eXchange of Indicator Information

TCP
Transmission Control Protocol

TTP
Tactics, Techniques, and Procedures

UDP
User Datagram Protocol

UIID
Universally Unique Identifier

URL
Uniform Resource Locator

VNF
Virtual Network Function

XML
Extensible Markup Language

XDR
Extended Detection and Response

YAML
YAML Ain't Markup Language

ZT-SDN
Zero Trust Software-Defined Networking

Appendix A

User manual

This chapter provides a practical guide for running the developed tool. The chosen setup ensures reproducibility of the execution environment and simplifies dependency management. The tool has been tested on Window host, but it is advised to rely on the Docker-based workflow to avoid compatibility issues. Before running the tool, a few prerequisites are required. In particular, the system requires a working Docker installation (the Docker Engine is required, Docker Desktop is recommended) to run the containerized development environment.

A.1 Deployment options

The repository provides two different approaches to run the tool:

- **Development environment (Devcontainer):** Defined by the `Dockerfile.dev`, which is based on `mcr.microsoft.com/devcontainers/python:1-3.13` and extends it by installing all the required system and Python dependencies.
- **Compiled image:** Defined by the `Dockerfile.prod_compiled`, which builds a self-contained executable version of the tool. This option is meant for running the tool without the need for the development environment.

A.1.1 Steps (Devcontainer)

1. Clone the source repository from GitHub onto the host machine.
2. Open the repository folder with Visual Studio Code.
3. When prompted, reopen the project in the `devcontainer` environment.
4. Wait for the Docker image to be built and the development container to start.
5. Once inside the container, the tool can be executed with:

```
python3 source/pb_rem.py ./tests/[folder_name]
```

A.1.2 Steps (Compiled image)

In addition, the repository also contains a `Dockerfile.prod_compiled` which allows building and running a compiled version of the tool. To use it:

1. Navigate to the project directory.
2. Build the image with:

   ```
   docker build -f Dockerfile.prod_compiled -t pb_rem:latest .
   ```

3. Run the container with:

   ```
   docker run --rm pb_rem:latest ./tests/[folder_name]
   ```

A.2 Native deployment (optional)

Although not recommended, the tool can also be executed natively on a host machine. In this case it is required to manually install:

- Python $\geq$ 3.13,
- Java Runtime Environment (OpenJDK 17),
- all Python dependencies listed in `requirements.txt`.

A Python virtual environment (`venv`) is highly suggested to avoid conflicts. After installation, the tool can be run from the project root as:

```
python3 source/pb_rem.py ./tests/[folder_name]
```

A.3 Cluster connection

If the user want to use cloud feature, the tool requires access to a Kubernetes cluster. The connection is handled through certificates mounted inside the `devcontainer`. This is achieved with the following lines inside the `devcontainer.json` file:

```
1  "mounts": [
```

```
    "source=C:/Users/${env:USERNAME}/.minikube/ca.crt,target=/workspaces
    /playbook-driven-remediator/config/k8s/ca.crt,type=bind,consistency=
    cached",
    "source=C:/Users/${env:USERNAME}/.minikube/profiles/minikube/client.
    crt,target=/workspaces/playbook-driven-remediator/config/k8s/client.
    crt,type=bind,consistency=cached",
    "source=C:/Users/${env:USERNAME}/.minikube/profiles/minikube/client.
    key,target=/workspaces/playbook-driven-remediator/config/k8s/client.
    key,type=bind,consistency=cached"
]
```

If correctly configured, the tool will automatically authenticate to the cluster and the user has only to insert port number in the file `source/settings.py`

A.3.1 Manual connection between cluster and Kubernetes API

If the automatic procedure does not work, this manual fix could be tried:

Open the file `source/helpers/kb_auth.py`, uncomment the first implementation of `get_certificate` function and set manually certificates (ca.crt, client.crt, client.key) of the cluster API server in the folder `source/helpers/kubernetes-certificates`. The correct configuration can be retrieved with:

```
kubectl config view
```

For reproducibility, a detailed step-by-step guide describing how to deploy the Kubernetes cluster used throughout this thesis is included in the project repository.

Appendix B

Developer manual

This chapter provides guidelines for developers who wish to extend or modify the framework.

B.1 Alerts and Threat Intelligence Updates

The entry point of the framework is represented by the alerts. Alerts can be freely added by developers, provided that they are organized in a dictionary-based structure.

When new alert types are introduced, it is necessary to update the threat intelligence mappings. In particular:

- If the alert corresponds to a new MITRE ATT&CK technique, the following files must be updated to ensure the correct mapping from MITRE code to CACAO playbook:
  - `attack_techniques.json`
  - `defensive_categories.json`
  - `defensive_technique.json`
  - `defensive_action.json`
- If the alert instead introduces new threat labels, these should be added or updated in the `threats.json` file.

For new threat categories, it will then be necessary to update the rem_info.json file with the useful information to map the alert in the global_scope. For the integration of new CACAO playbooks, the requirement is that the playbook is fully *CACAO compliant*. Furthermore, when adding new actions, a `step_extension` must be provided in order to declare additional parameters and execution logic if the action is not already implemented. A typical extension follows the structure:

```
"step_extensions": {
  "extension-definition--uuid": {
    "x_execution_function": "id_func",
```

```
    "name": "name_action_func",
    "description": "description of the action func"
  }
}
```

Finally, when developers want to extend the system with new recipes, these must comply with the grammar specified in the `grammar.tx` file, ensuring full compatibility with the recipe interpreter.

B.2 Extending KubeArmor and Cilium Policies

The framework provides support for the generation of KubeArmor and Cilium policies by leveraging a templating mechanism. Policies are defined using a `YAML`-based representation, and parametrization is achieved through the use of the `Jinja2` templating engine.

In practice, a policy template is defined with placeholders (e.g., `{{NAMESPACE}}`, `{{POD_LABEL}}`, or `{{NETWORK_PATH}}`) that are dynamically filled at runtime.

The filling process is performed programmatically using the Python `jinja2.Template` class. At runtime, the developer passes a set of parameters that replace the placeholders inside the policy template, producing a complete and valid YAML specification ready to be applied to the Kubernetes cluster.

B.3 Graph-Based Logical Modifications

The tool supports logical modifications not only at the enforcement level (e.g., network policies), but also directly on the graph representation of the system. This approach allows developers to model and apply transformations on the topology itself, ensuring a higher-level view of the interactions between nodes and services.

B.3.1 On-premises Service Graph

In the on-premises scenario, logical modifications are handled in the `service_graph.py` module. This file defines both the configuration of the network topology (including the types of nodes and their interconnections) and the set of low-level actions that can be applied to the graph. Examples of such actions include moving a node, adding or removing a firewall, or altering the connectivity between two components.

B.3.2 Kubernetes Cluster Representation

For the Kubernetes-based environment, a consistency with the service graph abstraction is maintained. Instead of a dynamic graph structure, the equivalent representation is provided by a `JSON` file. This file lists the pods in the cluster, along with their network properties, restrictions, dependencies, libraries, and potential artifacts related to security frameworks (e.g., ATT&CK techniques or D3FEND countermeasures).

The JSON abstraction allows reasoning about pod-level dependencies and potential security implications. The graph-like abstraction is primarily used for consistency and potential extensions, while the majority of security actions are enforced directly on the cluster.

B.4 Extending actions

The executor is organized into three main modules:

- `executor.py`: the main entry point of the executor, which coordinates the execution of actions and must be updated whenever a new action is implemented;
- `actions_cloud.py`: contains the list of cloud-related actions, which are executed on a Kubernetes cluster and often involve generating and applying security cilium policies
- `on_premises_actions.py`: contains the list of on-premises actions, which are typically derived from the `.rec` files. These actions may include the generation of KubeArmor policies, even though their enforcement is not directly supported.

For each new action, developers are required to:

1. Implement the action in the appropriate module *actions-cloud.py* or *on-premises-actions.py*;
2. Update `executor.py` by adding the corresponding method call;
3. If the action belongs to a new CACAO playbook, assign it the unique identifier setted in the CACAO playbook.

The cloud actions follow a common template, which ensures consistency across the codebase. This template includes parameter extraction, Kubernetes API interaction, optional policy generation from templates, and safe update of the global execution scope.

Listing B.1: Action_kubernetes template

```
def new_action(self, parameters, global_scope):
    """
    action description
    """
    logger.debug("executing new action")
    parameters = fill_from_context_value(parameters, global_scope)

    out_key = next(key for key, value in parameters.items()
                   if isinstance(value, dict) and value.get("out") is
   True)
    with self.global_lock:
        global_scope[f"{out_key}"] = {"type": ""}
        global_scope[f"{out_key}"]["cacao_id"] = global_scope.get(
```

```
            "cacao_identifier", "unknown")

    parameters_list = list(parameters.values())
    first_parameter= parameters_list[0]
    second_parameter= parameters_list[1]
    ...

    kubernetes_api = get_kubernetes_api("type_of_api_needed")
    #if a policy is needed
    template = self.cilium_templates.get("new_function_template")
    if template is None:
        logger.warning("new_function_template not found.")
        return


    params = {
        "parameter_template_1": first_parameter_passed,
        "parameter_template_2": second_parameter_passed
    }
    policy_yaml = generate_policy(template, **params)
    logger.info(f"Generated Policy YAML:\n{policy_yaml}")

    try:
        policy_dict = yaml.safe_load(policy_yaml)
        kubernetes_api.create_namespaced_custom_object(
            group="",
            version="",
            ...
            body=policy_dict
        )
        logger.info(f"Policy applied successfully")
        with self.global_lock:
            global_scope[f"{out_key}"]["result_field"] = ...

    #if a policy is not needed
    try:
        result=calling_kubernetes_api(parameters)
        with self.global_lock:
            global_scope[f"{out_key}"]["result_field"] = result



    except client.exceptions.ApiException as e:
        logger.error(f"Error API Kubernetes: {e.status} - {e.body}")
        with self.global_lock:
            global_scope[f"{out_key}"]["result_field"] = "error with new
     function"


    update_output_parameters(parameters, out_key, global_scope[f"{
    out_key}"])
```

On-premises actions also follow a common pattern, ensuring consistency across the

entire code base. This pattern includes how parameters are extracted, agent separation (e.g., container or network security), and how service_graph or iptables_rule_generator calls are handled.

Listing B.2: action on_premises template

```
def new_action(self, parameters, global_scope):
    """
    action description
    """

    first_parameter = parameters["name_parameter_1"]
    second_parameter = parameters["name_parameter_2"]

    if self.agent == 'container-security':
        template = self.kubearmor_templates.get("kubearmor_template.yaml
    ")
        if template is not None:
            params = {
                "param1": first_parameter,
                "param2": second_parameter
            }
            policy_yaml = generate_policy(template, **params)
            print(policy_yaml)

    elif self.agent == 'network-security':
        rule = parameters.get("name_parameter_rule", "iptables ...")

        if rule["level"] == 4:
            generated_rule = self.iptables_gen.generateRule("
    level_4_filtering", rule)
            print(generated_rule)

        try:
            self.graph.move(parameters)
        except Exception as ex:
            raise ex

    else:
        logger.error("agent not valid")
```