

Politecnico di Torino

Politecnico di Torino

Cybersecurity LM-32(DM270)

A.a. 2025/2026

Graduation Session September 2026

Development of a Threat Modeling Framework for SOAR Platforms

Supervisor:
Fulvio Valenza
Daniele Bringhenti

Candidate:
Alex Cugliari

Abstract

Security in a modern network cannot be simply reduced to the sum of how well each service defends itself. A modern Security Operations Centre is not a single tool but can be better described as a chain of interconnected components, each responsible for one stage of detection and response, and a problem introduced at a single point does not necessarily stay local : it can amplify and propagate along the dependencies that interconnect these components, reaching services that were never directly targeted and that share no visible relation with the one actually compromised.

A Security Orchestration, Automation and Response (SOAR) platform sits at the center of this architecture, correlating alerts and triggering remediation through playbooks that can act without waiting for a human approval step. This closes the loop between detection and response at machine speed, but it also means the automation's own decisions become one more part of what can go wrong. The dependencies that connect services to each other, and the dependencies that connect defensive components among themselves, form an attack surface that stays invisible if any single component is considered on its own, and becomes visible only once several parts of the pipeline are examined together.

Whether that last step happens with or without a human present in the loop is a choice with two structurally different failure modes. A playbook with no human gate reacts fast enough to matter against a threat that completes in seconds, but on the other hand, it propagates a mistaken one just as fast, since nothing downstream is built to question what it is given. A playbook gated behind the analyst's approval catches that specific fabrication, but the analyst is a single, shared, exhaustible resource, and every other gated playbook waits behind whichever one currently holds their attention. The circuit breaker added to catch a bad decision becomes, under enough load, indistinguishable from the chokepoint it was meant to guard against.

This thesis starts from the idea that human analysts need a tool that helps them reason about this workload systematically, rather than relying on a manual, case by case inventory of scenarios that stops scaling long before a realistic network does. It builds a formal model of a SOAR-protected network, one that treats the automation layer as part of the system under analysis rather than as an external, trustworthy observer of it, and operationalizes that model into a working tool, validated against a representative case study and relative queries, so that these interactions can be reasoned about with the same rigor as the network they want to protect.

Table of Contents

List of Figures III

1 Introduction 1
- 1.1 The Modern Network Security Operations Landscape 1
- 1.2 The Problem 3
- 1.3 Scope of the Thesis 4

2 Background: Network Security Automation 5
- 2.1 Why Manual Configuration No Longer Scales 5
- 2.2 Automating for Speed 6
- 2.3 Automating with Oversight 6
- 2.4 Automation's Expanding Reach 7

3 Background: Threat Modeling and Analysis 8
- 3.1 Foundations 8
- 3.2 Three Ways to Model a Threat 9
- 3.3 Checklists to Reachability: Attack Graphs 9
- 3.4 TAMELESS: A Hybrid Threat Model for Smart Systems 10
- 3.5 Positioning This Thesis 10

4 Thesis Objectives 12
- 4.1 Problem Statement 12
- 4.2 Objectives 13
- 4.3 Contributions 14

5 Analysis of the SOAR Ecosystem and its Dependencies 16
- 5.1 Two Motivating Scenarios 16
  - 5.1.1 The Cost of No Gate: Fast Propagation of a Bad Trigger . . 17
  - 5.1.2 The Cost of a Gate: The Analyst as a Shared, Exhaustible Resource 18
- 5.2 A Reference Architecture for SOAR-Protected Networks 19

5.3 Dependencies as a Hidden Attack Surface . . . 19
5.4 Case Study Topology . . . 20
5.5 Service Dependencies in the Case Study . . . 22
5.6 Amplification in Practice . . . 22

6 Threat Model 25
6.1 Attacker Model and Scope . . . 25
6.1.1 Attacker Capabilities . . . 25
6.1.2 Modeling Limitations and Design Choices . . . 26
6.2 Notation and System Sets . . . 26
6.3 Network Axioms and Structural Derivations . . . 29
6.4 Security State and Defense Predicates . . . 33
6.5 Derivation Rules . . . 35
6.5.1 Malfunction and Availability Loss . . . 35
6.5.2 Compromise and Lateral Movement . . . 37
6.5.3 Detection and Alert Fatigue . . . 38
6.5.4 Active Defense and Global Blacklisting . . . 41
6.6 Modeling SOAR Manipulation . . . 42

7 Implementation and Validation 44
7.1 Implementation Overview . . . 44
7.2 Validation Methodology . . . 44
7.3 Case Study Results . . . 46
7.3.1 Attack Surface Highlights . . . 47
7.3.2 Root-Cause Diagnosis . . . 49
7.3.3 Defense Comparison . . . 49
7.3.4 Crash Simulation: The Chokepoint Revisited . . . 51
7.3.5 Playbook Risk . . . 54
7.3.6 Attacker's View Against Defender's View . . . 54
7.3.7 SOAR Manipulation . . . 55

8 Conclusions and Future Work 56
8.1 Summary of Contributions . . . 56
8.2 Limitations . . . 58
8.3 Future Work . . . 58

Bibliography 60

List of Figures

1.1 The SOAR closed loop, with the branch that a playbook requiring human approval introduces: if no approval is required, remediation proceeds immediately; if it is required and the analyst is available, remediation proceeds after approval; if the analyst is occupied elsewhere, the playbook waits. 2

5.1 Physical topology of the case study network. Grey: external network. Coral: actuators (firewalls). Purple: sensor and orchestrator. Teal: ordinary assets. 21
5.2 The four logical connections of the case study, each modeled as an intermediate node representing the connection itself, to keep the dependency relation (Figure 5.3) structurally separate from the entities it relates. 22
5.3 Dependency chains of the case study. Each dependency is directed toward the connection it relies on, and from there toward the entity at the other end, never directly between the two entities, so that a break in the connection itself, and not only a failure of the remote entity, can be represented explicitly. 23

6.1 A physical link: symmetric, declared once, holds in both directions. 29
6.2 Whitelist: node fw_int locally authorizes traffic declared as originating from dns. 30
6.3 Has_Whitelist: a node with no configured rule falls back to default-allow. 30
6.4 Accepts: a node forwards traffic from S either by being bypassed through compromise, or through ordinary, unblocked routing. 31
6.5 A Path: a loop-free chain of physical links in which every intermediate node accepts traffic from S. 32
6.6 Logical_connection: two valid unidirectional paths, one in each direction. 32
6.7 Depends_on connection of [Entity → Relation → Entity]. 33

6.8 Malfunction propagation (entity cascade): a fault in B (orange, a state of disruption) is inherited by A through the dependency alone, not through any compromise of A. 36
6.9 Pivoting: full control over B, plus a usable vector, is enough to compromise A. 38
6.10 Detection: the sensor sits on the attacker's trace, and, independently, delivers an alert to the SOAR only if its own logical connection to the orchestrator is up. 39
6.11 Indirect detection: `ids` never appears in the attack path shown to the analyst, but removing it would break the logical connection the path relies on, so it still detects the traffic passing through it. . . . 40
6.12 Reaction: the SOAR updates the blacklist, the actuator blocks the attacker's traffic (marked), and any traffic further downstream toward the protected host remains inhibited as well. 42

7.1 The case study topology used for the query battery. Coral: entry points under attacker control. Orange: actuators, the chokepoints of Chapter 5. Purple: sensor and orchestrator. Teal: protected assets. Single arrows are physical links; double arrows are logical connections, drawn bidirectional here since only their existence matters. Every trace in the rest of this section is read against this map. 46
7.2 **Case 1: an external attacker, detected.** An attacker enters through `internet`, compromises `web_app`, and reaches `db` over the logical connection `conn_db_webapp` (single arrows are physical links; the double arrow is the logical connection, drawn in the direction this trace uses it; the thin gray dashed lines are not a connection at all, just an illustration of the hidden physical route). The printed path never names `ids`, but it is an unavoidable waypoint for that connection's physical route, so it detects the trace anyway (Section 6.5.3, Indirect Detection via a Hidden Chokepoint). Worst case, the SOAR reacts by blocking both `web_app` and `db`. 48
7.3 **Case 2: an insider, undetected.** The same asset, `db`, reached by the insider entry point instead: `host1` to `fw_int` to `db`, two physical hops (both solid arrows), no logical connection involved. `ids` sits nowhere on this route, so nothing reports it: `NOT DETECTED`. 48
7.4 **Example 1: two causes, one outcome.** Blocking `db` makes `web_app` lose availability through both the entity cascade (`db` is itself malfunctioning, shown by its own state) and the connection cascade (`conn_db_webapp` no longer holds under policy, shown by the broken double arrow) at once. Harmless redundancy, not a contradiction: the two rules agree. 49

7.5 **Example: quarantining the compromised device.** With `block( host1 )` in place, every other node reachable from `host1` becomes unreachable all at once: `fw_int` is `host1`'s only physical neighbor, so quarantining the source itself is enough to break every downstream path in one action. 50
7.6 **Example 1: the external attacker's perspective.** Crash of `fw_int` seen from `internet`. `fw_p`, `web_app`, and `ids` sit on the near side of the crashed node and stay reachable: `web_app` connects directly to `fw_p`, never crossing `fw_int` at all. `db`, `dns`, `host2`, and `soar` do not: the only physical neighbor of the `soar` is `fw_int` too, so its logical connection to `ids` breaks along with every other device, cutting the orchestrator off from its own sensor . Four nodes lose availability regardless, through the dependency chains of Chapter 5. 52
7.7 **Example 2: the insider's perspective.** The same crash seen from the case of `host1`. `fw_int` is `host1`'s only physical neighbor, so nothing remains reachable at all, not even `web_app`, whose only path from `host1` runs through `fw_int` and the sensor `ids`, nor `soar`, whose logical connection to `ids` is severed in the same way it was from `internet`. The lost dependencies are identical either way, since they depend on the topology of logical connections and not on who specifically caused the crash: the same single-chokepoint amplification of Section 5.3, here triggered by a real fault rather than a defense. 53
7.8 **The same question, two different answers.** Both predicates were run against the same hypothetical state: `ids` blacklisted, `fw_p` itself compromised with `Control` impact. The attacker's view accepts, because a compromised `fw_p` bypasses its own policy entirely; the defender's view rejects, because from policy alone the block looks like it worked. 54
7.9 **Three routes to the same loss, none of them a compromise.** `asset_threat_surface(web_app)` reports all three: a fabricated alert can name `web_app` directly, or name a node it depends on at any depth in the chain, and the connection-cascade rule of Chapter 6 carries the resulting block the rest of the way to `web_app`. `dns`, `db`, and `web_app` itself are never compromised in any of the three; the SOAR's own trust in what a sensor reports is the entire attack surface. 55

Chapter 1

Introduction

1.1 The Modern Network Security Operations Landscape

Nowadays, a modern Security Operations Centre (SOC) no longer consists of a single tool, but rather a chain of interconnected components, each of which is responsible and/or jointly responsible for one or more different stages of detection and response. At the perimeter level, sensors, which can be described as systems that detect actions inconsistent with normal network operation (such as IDS, or **Intrusion Detection Systems**), analyse network traffic for known signatures and anomalous patterns, generating raw logs and alerts. These raw signals converge in **Security Information and Event Management** (SIEM) platforms, which aggregate and normalise log data from heterogeneous sources, such as network sensors, **endpoint detection and response** (EDR) agents, firewalls and threat intelligence feeds. They also correlate this data into structured alerts classified by priority. An EDR layer complements this by directly monitoring endpoints, providing host-level telemetry data (process creation, registry access and the like) that a purely network-based sensor is unable to detect, and acting as one of the operational channels through which remediation is ultimately carried out.

The component that in general sits at the centre of this pipeline is a **Security Orchestration, Automation and Response** (SOAR) platform. A SOAR receives the correlated alerts produced by the SIEM, or other equivalent system, and evaluates them against a library of automated response procedures also known and called as playbooks, in fact, when a playbook's trigger and conditions are met, it will issue the corresponding remediation actions through a specific class of nodes present in the network called the **actuators**. An example of actuators is:

- FIREWALLS follow a set of rules (called whitelist or Allowlist) that block a malicious source;

- IDENTITY PROVIDERS that are systems with the ability to disable the access rights of a compromised account;
- ENDPOINT AGENTS that have the ability to isolate a specific host or program through the use of different tools.

In theory, a SOAR platform is typically organised into three main functional layers: The first one is an integration layer, which uses APIs and webhooks to connect different (or better heterogeneous) external tools; a workflow engine, which executes playbooks, evaluates their conditional branches, and turns raw events into concrete sequences of actions; and as a third layer there is the case management, which logs the resulting incidents and, if it is specifically configured, gates specific actions behind human approval.

This architecture is commonly described as a closed loop: "detection -> correlation -> decision -> remediation" form a continuous cycle that, for the playbooks that are configured to require no human step, once it is correctly configured, does not require further human intervention for the categories of incident it is designed to handle. Not every playbook takes this path, however: Figure 1.1 also shows the branch a playbook takes when human approval is required instead, and what happens to it while the analyst who must grant that approval is occupied elsewhere.

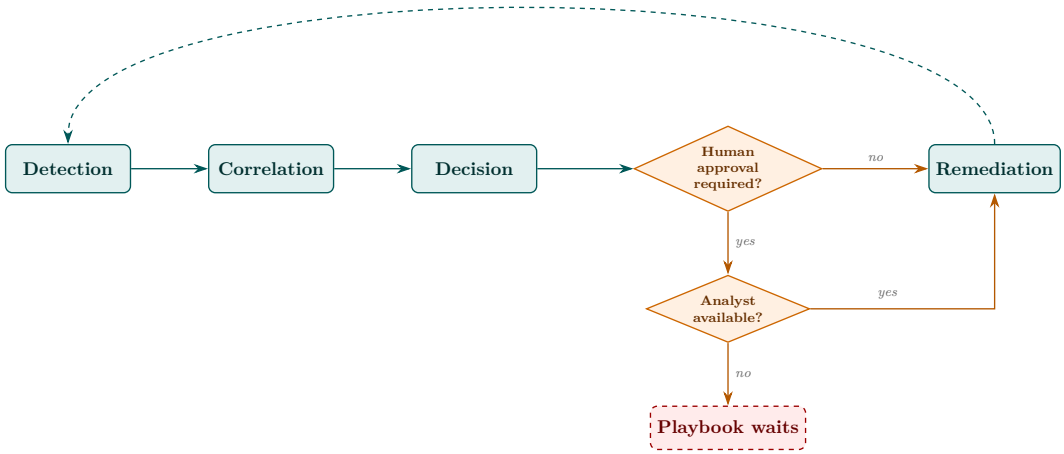


Figure 1.1: The SOAR closed loop, with the branch that a playbook requiring human approval introduces: if no approval is required, remediation proceeds immediately; if it is required and the analyst is available, remediation proceeds after approval; if the analyst is occupied elsewhere, the playbook waits.

A pattern of this kind has already been demonstrated concretely, for the specific case of IDS-triggered firewall reconfiguration, by fully automated approaches such

as the one proposed and described in papers such as "Autonomous attack mitigation through firewall reconfiguration" . [1]

Industry sources generally tend to report that this closed-loop automation reduces the mean time to respond to an incident and allows a fixed-size SOC team to absorb a growing volume of alerts without a proportional increase in staff, at the not always negligible cost of the variability that comes with individual human judgment [2].
The interaction of these many components, each one that is contributing with its own data, its own decisions, and its own specific modes, generates a system whose overall behaviour is difficult for a single analyst to imagine and reason about by hand, in other words, the number of ways a threat can propagate, or a defensive action can interact with a dependency it was not designed to consider, grows exponentially faster than the number of components involved. This intrinsic complexity is precisely why tools that support human analysts are needed. Those tools help in identifying and reasoning about the risks and threats such a system faces.

The security world continues to move in this direction: industry analysts have in recent years begun to question how long SOAR platforms in their current form will remain the dominant paradigm, as attention shifts toward more autonomous, and in some cases partially AI-driven, security operations tools. If anything, this trend sharpens rather than weakens the motivation for this thesis, since a more autonomous pipeline leaves even less room for a human circuit breaker.

1.2 The Problem

The pipeline described in Section 1.1 is not a single design choice but is better to consider it as a possible structure, and the SOC architect who configures it is, whether is explicitly conscious about it or not, always choosing a position along that spectrum for every playbook created. At one end, a playbook executes with no human step at all: this is what makes the loop of Figure 1.1 closed, and it is what makes the response fast enough to matter against threats that can be quick in completing action in the range of seconds. At the other end, a playbook's most consequential actions are gated behind an analyst's explicit approval, precisely because removing that gate is what makes the fast and uncontrolled path of responces dangerous in the first place. Neither position is safe by itself, and, more importantly for this thesis, each is unsafe for a structurally different reason.

A playbook with no human gate propagates whatever it is given, genuine or fabricated, at machine speed and without a second opinion. On other hand, a playbook with a human gate is only as available as, and as fast as, the analyst

behind it, and that analyst is a single, shared, and exhaustible resource. In fact an incident that occupies them for an extended period of time and not only for the few seconds that a common routine approval takes, for example like for the hours a genuine ransomware response can require, leaves every other gated playbook unable to execute for as long as the occupation and the task lasts. The gate that was added to prevent one failure mode becomes, under sufficient load, the mechanism of a different one, in other word a **BottleNeck**.

This is the more general problem this thesis is concerned with: the interaction of many components, some automated and some human, produces failure modes that neither the automated components nor the human ones exhibit on their own, and that are easy to miss precisely because they only appear once several parts of the pipeline are considered together rather than in isolation. Reasoning about this by hand does not scale much further than the two examples above; a network with a realistic number of playbooks, sensors, and dependency chains has far too many ways for a fast, ungated path and a slow, gated one to interact for a human analyst to enumerate them all with confidence.

1.3 Scope of the Thesis

Given the complexity outlined above, this thesis is founded on the idea that human analysts need a tool that helps them reason about this workload systematically, rather than relying on a manual, case-by-case inventory of scenarios. This is the scope of the present work: a formal, automatically checkable model of the SOAR pipeline, together with the tool that operationalizes it, so that the interactions described in Section 1.2 can be reasoned about with the same rigor as the network they protect.

Chapter 2

Background: Network Security Automation

This chapter gives a general overview of the research on network security automation that this thesis is built on, paying particular attention to the academic literature closest to the approach developed later in this work. The same duality that runs through the rest of this thesis, in particular between automating a response for speed and keeping a human able to catch a bad decision, is already expressed through this literature. The chapter is organized around that duality: Section 2.2 covers work that pushes toward full automation, Section 2.3 covers work that deliberately keeps a human in the process, and Section 2.4 shows the same automation extending to new kinds of security function and new deployment paradigms with some examples and related references.

2.1 Why Manual Configuration No Longer Scales

The starting motivation for this entire research area is stated concisely in the survey *Automation for Network Security Configuration: State of the Art and Research Trends* [2]: the size and complexity of modern computer networks are increasing as a consequence of architectural paradigms such as the Internet of Things and network virtualization, to the point that manual orchestration and configuration of network security functions is no longer viable in an environment where a single minimal configuration error can be exploited by an attacker. The survey identifies and explains Software-Defined Networking and Network Functions Virtualization as the key technological enablers that make automation both necessary and possible: by separating network functions from dedicated hardware and by centralizing their control, these paradigms make it possible to achieve and feasible to compute a security configuration using programs rather than by hand, and to deploy again it

as the network changes.

This is the motivation this thesis inherits, and it already contains the foundations of the duality that structures this chapter. If configuration is now automated and reactive rather than manually curated, the automation layer inherits the trust that used to be left in the hands of a human administrator. Whether that trust is well placed depends on the balance of how much of the decision the automation is allowed to make on its own, which is exactly the design choice the rest of this chapter aims to track across the literature.

2.2 Automating for Speed

Firewall placement and configuration is the area where full automation has produced the strongest formal guarantees: a configuration can be computed with a proof that it satisfies the stated requirements and uses the smallest number of firewalls needed, which is enough on its own to deploy it without a human checking it by hand, and the same correctness-by-construction approach has since been extended to further objectives, from minimizing firewall count to optimizing for energy consumption [3, 4, 5].

That commitment is pushed furthest by work that moves from static configuration to reactive configuration, automatically reconfiguring a firewall the moment an IDS raises an alert, with no human step between detection and remediation [1]. Because this pipeline is validated as fully automated, it exposes a gap this thesis is concerned with: the alert that triggers it is treated as ground truth, and what happens if that alert was never genuine to begin with, whether because the sensor itself has been compromised or because an attacker found another way to inject a fabricated entry, is not part of what is modeled. This scenario is picked up again and formalized later in this thesis.

2.3 Automating with Oversight

Not every paper in this line of work makes the same choice. Work on firewall anomaly resolution has argued explicitly against full automation, since an automated fix to a conflicting pair of rules may produce a result the administrator never intended: sub-optimizations with an unambiguous fix are still resolved automatically, but a conflict that requires a judgment call is instead routed through explicit queries to the administrator [6, 7]. This is a different point on the same spectrum discussed in Chapter 1: the tool is still doing most of the work, but the actions with the least reversible consequences are deliberately routed through a human, and anomaly resolution, unlike the allocation problem of Section 2.2, has consistently been

treated in this literature as a task where full automation is not automatically the right default.

2.4 Automation's Expanding Reach

The methodologies described so far are not confined to firewalls. The same correct-by-construction guarantee carries over from packet filters to Web Application Firewalls, reasoning about HTTP semantics rather than IP addresses and ports [8], and a similarly automated, policy-based approach has been applied to network isolation, first for Kubernetes and then across the wider edge-to-cloud continuum [9, 10].

Both directions point in the same broad sense: automation absorbing responsibilities that used to require a human administrator is not a property of firewalls specifically. SOAR-driven incident response, the specific case this thesis is concerned with, is a further instance of the same trend, and it inherits the same open question already visible in Section 2.2 and Section 2.3: whether the human is removed from the loop, or kept exactly where the consequences of a mistake are hardest to undo.

Chapter 3

Background: Threat Modeling and Analysis

This chapter introduces the concepts and methodologies that the framework developed later in this thesis builds on, drawing on the terminology and taxonomy taught in the Threat Intelligence course at Politecnico di Torino [11] and on the hybrid threat model proposed in *A Hybrid Threat Model for Smart Systems* [12], which is the closest methodological precedent to this thesis and the direct source of its notation. The chapter is meant to work as a bridge: it fixes the vocabulary and the landscape of existing approaches first, then places the contribution of this thesis inside that landscape rather than next to it.

3.1 Foundations

A threat model needs only a handful of terms to get started: an asset is anything that has value and therefore requires protection, a threat is a potential cause of harm to that asset, a vulnerability is a weakness a threat can exploit, and a compromise is what happens once it does [11]. This is the same vocabulary the formal model of this thesis is built on later: entities are assets, and the assumed and derived properties introduced further ahead track exactly the transition from a vulnerability that exists to a compromise that has occurred.

Understanding where a system is exposed, independently of how those exposures are later scored or prioritized, already has value on its own [11]. This is the part of the process this thesis is concerned with: deriving whether an attack path or a manipulation of the automation layer is possible at all, without attempting to rank the resulting risks.

3.2 Three Ways to Model a Threat

No single methodology dominates threat modeling, and the three most established ones start from different places. STRIDE is design-centric, checking each component of a system against six categories of violation; MITRE ATT&CK is adversary-centric, cataloging how real attackers actually operate rather than starting from a system description at all; and PASTA is risk-centric, folding threat modeling into a multi-stage process that ends in a formal risk evaluation [13, 14, 15]. None of the three reasons about a system through the paths an attacker can take across it, which is the starting point this thesis needs, and the next section introduces the family of approaches that does.

3.3 Checklists to Reachability: Attack Graphs

Monitoring tools and the methodologies above share a common limitation: they reason locally, evaluating individual components or weaknesses in isolation. A vulnerability, however, has no meaning only by itself. Its risk depends on reachable paths and on the other weaknesses that it can be combined with. In fact, the question that really matters is not whether or not a component is vulnerable but whether there is a path from the attacker to a critical asset: security, in this particular framing, is a reachability property, not a checklist property [11].

Attack graphs formalize this shift, and the family most relevant here is the condition-based, or logical, one: instead of complete system states, nodes represent facts and exploits, and the graph is bipartite, $G = (C \cup E, \rightarrow)$, with an edge from a condition to an exploit marking a precondition and an edge back marking a postcondition [11]. Once a condition becomes true it stays true, and new conditions accumulate on top of it, which avoids the state explosion of a model built on complete system states.

The framework developed later in this thesis is, in this taxonomy, a condition-based attack graph, generated in what this taxonomy calls an analysis-driven way: it evaluates a set of logical rules against a declarative network description, rather than following fixed transformation rules or matching known patterns [11]. The best known example of this generation strategy is MulVAL, which does the same over a Datalog specification rather than a Prolog one [16, 17].

Two ways of querying such a graph are used later in this thesis. Forward chaining starts from known true conditions and computes everything reachable from them, and is well suited to blast-radius questions such as what becomes reachable if the attacker already holds a given foothold. Backward chaining starts instead from a goal and expands only the prerequisites that support it, and is well suited to asking whether a specific target is reachable at all.

3.4 TAMELESS: A Hybrid Threat Model for Smart Systems

The threat model proposed in [12] is a condition-based attack graph in the sense above, specialized to cyber-physical systems, and it is the direct source of the notational convention adopted later in this thesis: assumed properties, prefixed α, versus derived properties, prefixed κ.

Its central argument is that existing threat models fail to capture hybrid attacks: multi-step attacks that combine exploitation of the human, physical, and cyber aspects of a system in any order, propagating across the interfaces between them. To represent this, it defines entities that can be of cyber, physical, or human nature, relations between them such as *Contain*, *Depend*, and *Monitor*, and properties such as *Compromised*, *Malfunctioned*, and *Vulnerable* that describe an entity's evolving state. As in this thesis, assumed properties must be declared as input while derived properties are computed automatically by a set of derivation rules, and the model is implemented on top of an XSB Prolog interpreter, the same tool this thesis relies on.

3.5 Positioning This Thesis

The framework developed later in this thesis sits at the intersection of the two traditions this chapter has covered. From the attack-graph literature of Section 3.3 it inherits the basic commitment: security is treated as a reachability property over conditions and exploits, generated in an analysis-driven way and queried in both its forward and backward form. From TAMELESS it inherits the assumed-versus-derived notation and the idea that a threat model should represent the specific domain it targets rather than a generic system.

The specific domain targeted here, in other words a SOAR protected network, is where this thesis extends both traditions rather than only combining them. In fact, neither the general attack-graph literature nor TAMELESS represents the particular case where the entity that detects and reacts to a threat is not an external observer, but a component with its own attack surface. TAMELESS's *Monitor* relation, in particular, treats the monitoring entity as a source of ground truth: if an entity monitors another for a given threat and is itself valid, the compromise can be detected. What it does not represent is a monitoring entity that has itself been compromised and, instead of merely failing to detect a real threat, actively fabricates a report about an uninvolved third entity. That mechanism, formalized later in this thesis together with the neutralization of a human-gated playbook by keeping its analyst occupied elsewhere, is the specific contribution this thesis adds to the landscape surveyed in this chapter: not a new family of attack graph,

but a new class of condition being modeled within a familiar one, one where the defense itself, on either side of the automation boundary discussed in Chapter 1, can become part of the attack surface it was built to close.

Chapter 4

Thesis Objectives

As outlined in Chapter 1, the growing reliance on SOAR platforms to automate incident response introduces a class of risk that is not adequately captured by threat models that treat the automated defense as an implicitly trustworthy, external observer of the network. The objective of this thesis is to close this gap by developing a threat modeling framework that treats the automation layer, in both its automated and its human-gated form, as part of the system under analysis, rather than as a component external to it.

4.1 Problem Statement

This thesis addresses the lack of a formal, systematic way to reason about the security of the automation layer of a SOAR-driven defense, jointly with the security of the network it protects. Three properties of the problem make it distinct from classical network threat modeling:

1. **The automation layer introduces a second attack surface that is causally coupled to the first.** An attacker's ability to manipulate the SOAR does not depend only on the SOAR's own vulnerabilities, but also on the state of the network it monitors: for instance, whether a sensor positioned on a given network segment has itself been compromised, or whether a specific logical connection to the orchestrator is active. Reasoning about one surface in isolation from the other misses attacks that exploit exactly this coupling.

2. **The consequences of a manipulated response are not confined to the node under attack.** Because services in a modern infrastructure depend on one another (a web application on a database, a database on a name resolution service, and so on), a defensive action that blocks or isolates one node can cause a cascading loss of availability in services that were never themselves

compromised. A threat model that only tracks direct compromise, without tracking these dependency chains, cannot predict this class of self-inflicted damage.

3. **The human component of the pipeline is itself a resource with finite capacity, and therefore a target.** A playbook gated behind an analyst's approval is not safe by default: its availability depends on that analyst not being occupied elsewhere, and an attacker, or an ordinary high-severity incident, can exhaust that availability without touching the gated playbook at all. A threat model that represents human approval only as a binary condition, present or absent, cannot capture the case where the gate exists but is unavailable exactly when it is needed.

A useful threat modeling framework for SOAR platforms must therefore represent, in a single formal model: the network topology and the dependencies between its services; the vulnerabilities of individual components and the conditions under which an attacker can pivot from one compromised node to the next; the behavior of the automated defense, including the conditions, automated or human, under which it reacts and the actions it is capable of taking; and the ways in which the defense itself, on either side of that condition, can become an instrument of attack rather than a countermeasure against it.

4.2 Objectives

More specifically, the thesis pursues the following objectives.

1. **Characterize the behavior of SOAR platforms and closed-loop remediation architectures** to the extent required to build a model that is faithful to how these systems operate in practice: how alerts are generated and correlated, how playbooks are triggered and executed, and where human approval gates, if present, act as control points in the response pipeline.

2. **Define a threat model tailored to SOAR platforms** that systematically represents the core architectural components of a SOAR-protected infrastructure and their interactions. The model must capture, at the bare minimum: the network topology and the physical and logical connections between its components and the dependencies between services, so that the consequences of a compromise or of a defensive action can be traced beyond the node they directly affect; the vulnerabilities of individual components and the conditions under which an attacker can use a compromised node as a pivot toward further targets; and the internal logic of the SOAR itself, including sensors, playbooks, and the conditions that enable, disable, or redirect an automated response.

3. **Implement a threat derivation engine using a rule-based approach.** Among the candidate technologies considered for this purpose, production-rule systems such as CLIPS and logic programming systems such as Prolog, this thesis adopts Prolog, on the basis of the close correspondence between its resolution mechanism and the first-order logic formalization used for the threat model, and of the direct support it provides for the exploratory, backtracking-based search over alternative attack paths that the model requires.

4. **Apply and validate the framework on a case study**, in the form of a representative simulated network topology, in order to assess its practical applicability. This includes verifying that the derivation engine produces results consistent with the formal model it implements, and using the case study to surface concrete instances of the risks described in Chapter 1, such as a defensive action that inadvertently damages the availability of a service it was not intended to affect, or a sensor whose compromise allows an attacker to redirect the SOAR's response toward an uninvolved target.

These four objectives correspond, respectively, to the content of Chapter 5 (objective 1, together with part of the analysis underlying objective 2), Chapter 6 (objective 2), and Chapter 7 (objectives 3 and 4).

4.3 Contributions

This thesis makes the following contributions:

- A **formal threat model** for SOAR-protected network infrastructures, expressed in first-order logic, that represents network topology, service dependencies expressed as logical connections, node vulnerabilities and their impact, and the reactive behavior of the SOAR platform, including sensors, playbooks, and the conditions under which a playbook can be triggered, neutralized, or manipulated into acting against an unintended target.

- A **rule-based derivation engine**, implemented in Prolog, that operationalizes the model: given a declarative description of a network (its topology, its services, its vulnerabilities, and its defensive posture), the engine derives attack paths, cascading availability losses, sensor detection outcomes, and the residual risk after a defensive action has been simulated.

- A **validation methodology and case study** that applies the framework to a representative network topology, and that combines manual verification, independent review by multiple human and AI-based readers, and comparison of outputs across engine revisions to establish confidence in the derivation

engine's correctness, including the identification and correction of a concrete defect in the engine's dependency-chain analysis, and the identification of a methodological limitation in how the framework currently compares an unrestricted attacker view against a purely policy-based view.

Chapter 5

Analysis of the SOAR Ecosystem and its Dependencies

This chapter presents the analysis of the SOAR ecosystem that motivates and grounds the threat model of Chapter 6. The central argument of this chapter is that the risks described in Chapter 1 cannot be understood, let alone modeled, by looking at individual components in isolation. What makes a SOAR-protected network fragile in ways that are easy to overlook is the *network of dependencies* that connects its components, both at the level of the services that depend on one another to function, and at the level of the defensive components that depend on one another to detect and react to a threat. A problem introduced at a single point in this network, whether by an attacker or by the automated defense itself, does not necessarily stay local: it can amplify as it propagates along these dependencies, affecting components that were never directly targeted.

5.1 Two Motivating Scenarios

Section 1.2 argued, in general terms, that a pipeline built from both automated and human-gated components fails in two structurally different ways, depending on which side of a given playbook is weaker: the automated side propagates a bad trigger as fast as a genuine one, and the human-gated side is only as available as the analyst behind it. This section makes each side concrete, before the rest of this chapter turns to the architecture and dependencies that make both scenarios possible.

5.1.1 The Cost of No Gate: Fast Propagation of a Bad Trigger

The case for the first failure mode rests on a single structural observation: every component in the pipeline of Section 1.1 trusts the component that feeds it, and none of them is, by construction, in a position to verify that trust. The trust, in particular, flows blindly upward from the sensor to the SIEM, then to the SOAR and after that finally to the actuator. In fact, because remediation is driven by data generated at the very bottom of this chain (for example: a packet, a log line, a JSON alert), an attacker who can influence that data does not need to compromise anything downstream. The corruption is inherited almost automatically, by the components that hold privileges the original data source never had, and, when we talk about the ungated path, it is acted upon just as quickly as a genuine alert would be.

To see how this works, let's consider an IDS that inspects packets without maintaining connection state, a common and often unavoidable trade-off of performance. Such a sensor has no way to check that the source in a packet's IP header is genuine, but only that the packet is syntactically correct. An attacker who can craft a spoofed packet, for example one whose source is set to that of the organization's own DNS server and whose payload matches exactly a signature the IDS is configured to flag, causes the IDS to raise an alert that puts the DNS server as the attacker rather than the real true source. Later in the chain the SIEM that receives this alert has no independent way to question it: its role is, in fact, to normalize and correlate and not to try again to verify the real packet provenance, so it forwards a structured, often with high-confidence, alert naming the DNS server as malicious. The SOAR, at the end, is not designed to check an alert it receives from a trusted SIEM: it extracts the offending address and instructs the firewall to block it. The attacker never touches the DNS server, the firewall or the SOAR itself, and yet the organization's own name resolution service ends up isolated by its own defenses: a denial of service inflicted entirely by the automation layer that was supposed to prevent it, and inflicted in the time it takes a playbook to run rather than in the time it would take a human to notice something was off.

This pattern generalizes beyond a single spoofed packet. If the integration layer of the SOAR exposes an API (or webhook endpoint), as it typically must in order to accept alerts from external threat-intelligence devices or general custom tooling, and that endpoint's credentials are exposed or simply guessable, an attacker can inject a fabricated and high-priority incident directly, bypassing the IDS and SIEM entirely. Due to the fact that playbooks in general iterate over attacker-influenced data as well, such as a list of IP addresses extracted from an email header, a sufficiently large or malformed input can separately drive the workflow engine into a resource-exhaustion state, degrading or disabling completely the very capability

meant to keep the response pipeline fast. Neither scenario requires a defect in the SOAR's implementation: each one exploits the pipeline as designed, by supplying it with data it has no mechanism to question, and each one can be only as dangerous as it is because nothing downstream of the sensor is built to slow it down.

5.1.2 The Cost of a Gate: The Analyst as a Shared, Exhaustible Resource

The opposite failure mode is easy to overlook precisely because it looks, on the surface, like the responsible design choice. Gating a playbook's most consequential actions, such as a global blacklist or a change to an identity provider, behind an analyst's explicit approval is a reasonable response to the risk described above: a human who has to click "approve" has, at least in principle, the chance to notice that the request does not make sense. That protection, however, is only as strong as the analyst's availability, and availability is not something a single person has in unlimited supply.

An analyst is not just occupied for the few seconds a routine approval should require. Some incidents, like a ransomware outbreak, require an analyst's continuous and sustained attention for hours: triaging affected hosts, coordinating containment and often communicating with stakeholders. For as long as that occupation lasts, every other playbook that is gated (can be in a waiting list or just blocked) behind the same analyst, or the same limited pool of analysts, cannot execute, regardless of how urgent it is. The gate does not fail by being bypassed. In fact, it fails by being unavailable exactly when something else needs it. This is not a hypothetical concern: it is a direct consequence of treating human approval as a shared resource with finite capacity, which is precisely what it is.

What makes this failure mode more than an operational problem is that it does not require a real incident to trigger it. An attacker who cannot compromise anything technical can still inject a fabricated and high-priority alert designed specifically to consume the analyst's attention, such as a report of ransomware on the CEO's laptop. The SOC concentrates its resources on the fabricated emergency, so every other gated playbook waits behind it; and the attacker proceeds, mostly undisturbed, against the actual target. The circuit breaker that was added to catch a bad decision becomes, under this kind of pressure, indistinguishable from a chokepoint: the same structural role that Section 5.3 identifies for a single node whose failure breaks several independent dependencies at once, except that here the node in question is the availability of the analyst, and an attacker does not need to compromise anything downstream of it to exploit it.

5.2 A Reference Architecture for SOAR-Protected Networks

Independently of the specific product or vendor, a SOAR-protected network can be described in terms of a small number of functional roles.

- **Assets** are the components that provide the actual business functionality of the network, such as application servers, databases, and name resolution services. Assets are the entities an attacker ultimately wants to compromise, and the entities whose availability the defense ultimately wants to preserve.
- **Actuators** are the components capable of enforcing a network-level policy decision, typically firewalls or, more generally, any device that can accept or reject traffic on the basis of a set of rules. In the context of automated defense, actuators are what a SOAR platform manipulates when it executes a containment action, for instance by adding a source to a global blacklist.
- **Sensors** are the components responsible for observing traffic and generating security-relevant events, typically an Intrusion Detection System (IDS) or similar monitoring infrastructure. A sensor's usefulness depends on two conditions that are easy to state but, as this thesis will show, not always simultaneously satisfied: it must actually be positioned along the path an attack takes, and it must itself be trustworthy at the time it generates an alert.
- **The orchestrator** (the SOAR platform itself, or an equivalent SIEM/SOC component) is the component that receives the alerts generated by sensors, evaluates them against a set of automated response procedures (playbooks), and, when a playbook's trigger and conditions are satisfied, issues the corresponding actions to the actuators.

A playbook can be described, at this level of abstraction, as a conditional rule with three parts: a *trigger*, describing the class of alert that activates it; a set of *conditions*, describing the operational prerequisites for its execution (for instance, that the orchestrator itself is not in a degraded state, or that a human analyst is available if the action requires approval); and a set of *actions*, describing the concrete effect of the playbook once triggered, such as instructing an actuator to block a given source.

5.3 Dependencies as a Hidden Attack Surface

Independently of any specific topology, two kinds of dependency need to be mapped in a SOAR-protected network. The first is the ordinary kind: one service depends

on another to function, a web application on its database, a database on the name resolution service it uses to reach other hosts. The second is easy to miss because it sits one level up, among the roles of Section 5.2 rather than among the assets they protect: a sensor depends on its connection to the orchestrator to deliver an alert, an actuator depends on receiving a command from the orchestrator to enforce it. Both kinds of dependency matter for the same reason: neither a compromise nor a defensive action stays confined to the component it directly touches.

A dependency, in the model this thesis uses, is never expressed directly between two entities. It always passes through the connection that realizes it: an entity depends on a specific logical connection, and that connection in turn depends on the entity at its other end. This is not a modeling technicality. It is what makes it possible to distinguish two situations a coarser model would conflate: an entity becoming unavailable because the remote service it relies on has failed, and an entity becoming unavailable because the connection between them can no longer be established at all, even though the remote service itself is perfectly healthy.

This distinction is what the analysis behind this thesis turned up as its central finding. When several of these connections are independent of one another, in the sense that none of them depends on any other, but their underlying physical routes happen to converge on the same chokepoint, a single failure at that chokepoint does not ripple through the network as a chain reaction. It produces multiple, simultaneous, and independent losses of availability, in services that share no relationship with one another and that were never themselves compromised. A defensive action that blocks or isolates that one chokepoint node, taken with no visibility into how many otherwise-unrelated dependencies pass through it, can therefore cause damage that looks, from the outside, like several unconnected incidents happening at once rather than the side effect of a single decision. Section 5.6 shows this pattern occurring concretely in the case study used throughout the rest of this thesis.

5.4 Case Study Topology

To ground the analysis, and to provide a concrete testbed for the framework developed in the rest of this thesis, a representative network topology was defined, instantiating the roles of Section 5.2. The topology, shown in Figure 5.1, is organized into three zones: a perimeter zone, separating the external network from the rest of the infrastructure through a perimeter firewall (`fw_p`) and a sensor (`ids`); an internal zone, hosting the protected assets behind an internal firewall (`fw_int`); and the orchestration component (`soar`), reachable from the sensor through a dedicated logical channel.

Two entities, `internet` and `host1`, are treated as potential entry points for an

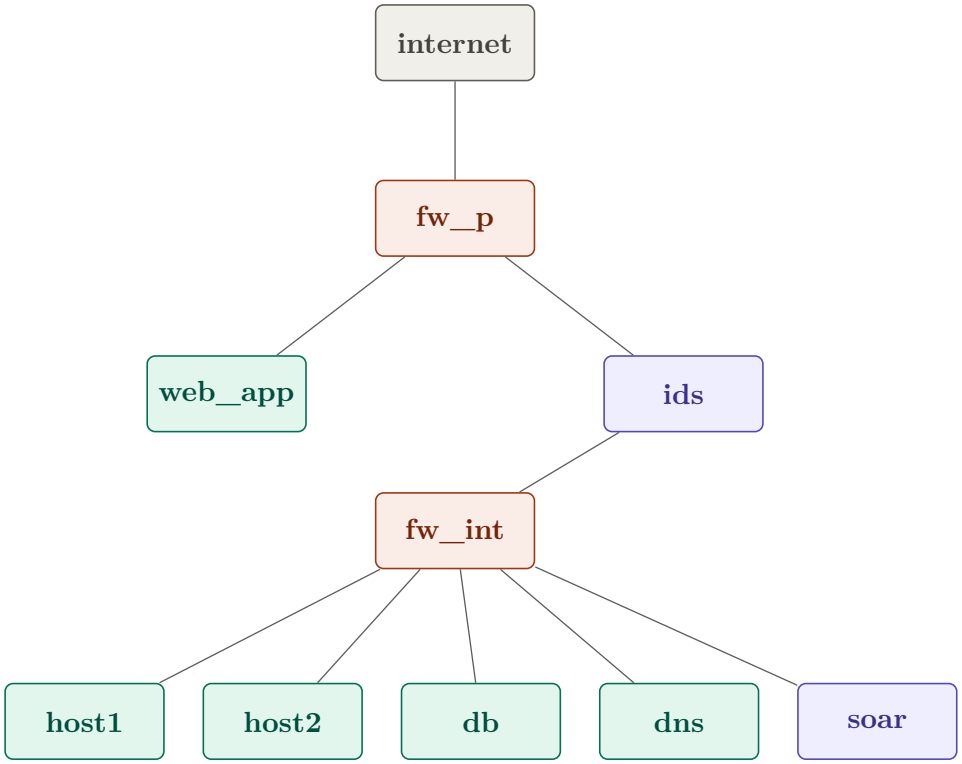


Figure 5.1: Physical topology of the case study network. Grey: external network. Coral: actuators (firewalls). Purple: sensor and orchestrator. Teal: ordinary assets.

attacker: the former models an untrusted external network, permanently outside the defender's control; the latter models an insider threat or an already-compromised endpoint. Neither entry point is included in the whitelist of any firewall along its path, so that its ability to reach the rest of the network depends entirely on exploiting a vulnerable node along the way, rather than on any explicit authorization.

5.5 Service Dependencies in the Case Study

Section 5.3 described, in general terms, why a dependency is modeled through the connection that realizes it rather than directly between two entities. This section shows what that looks like in the case study topology of Figure 5.1.

Four logical connections were defined for the case study, each representing an application-level interaction that abstracts away the specific protocol involved (an API call, a database query, and so on), and each requiring, structurally, that a valid path exists between the two endpoints in *both* directions:

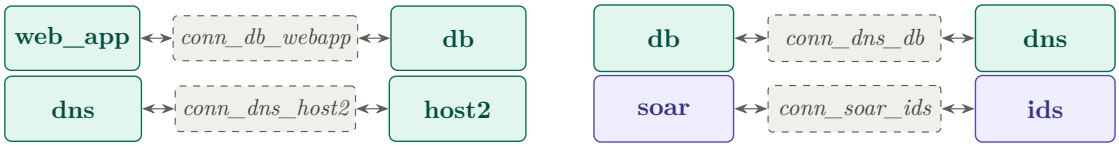


Figure 5.2: The four logical connections of the case study, each modeled as an intermediate node representing the connection itself, to keep the dependency relation (Figure 5.3) structurally separate from the entities it relates.

Building on these logical connections, a *depends-on* relation captures which entities require which connections, and, transitively, which other entities, to remain available. Figure 5.3 shows the dependency chains of the case study: `web_app` depends on its connection to `db`, which in turn depends on its connection to `dns`; `host2` depends directly on its connection to `dns`.

5.6 Amplification in Practice

Section 5.3 argued, in general terms, that a single failure at a chokepoint shared by several independent dependencies produces multiple, simultaneous losses of availability rather than a single chain reaction. This section confirms that the pattern is not merely possible in principle: it occurs in the case study topology itself, at the node that was placed there specifically to connect the internal zone to the rest of the network.

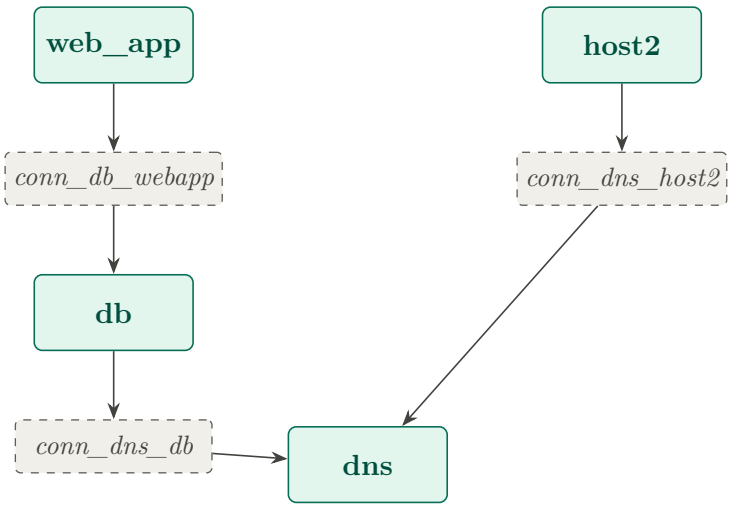


Figure 5.3: Dependency chains of the case study. Each dependency is directed toward the connection it relies on, and from there toward the entity at the other end, never directly between the two entities, so that a break in the connection itself, and not only a failure of the remote entity, can be represented explicitly.

In the case study, `fw_int` plays exactly this role. It is the only physical link between the internal zone and the rest of the network, and, as Figure 5.2 shows, all four logical connections of the topology have an underlying physical route that passes through it: three of them, `conn_db_webapp`, `conn_dns_db`, and `conn_dns_host2`, connect assets to one another, and the fourth, `conn_soar_ids`, connects the sensor to the orchestrator itself. If `fw_int` becomes unavailable, whether because of a real fault or because it has been isolated by a genuine and automated defensive action, all three connections are broken *simultaneously and also independently*: `db` loses its connection to `dns`, the same goes for `host2` that loses its connection to `dns` and `web_app` that loses its connection to `db`. Three assets that at first sight share no direct relationship with one another, and none of which was itself specifically compromised, lose availability as a side effect of a cascade from a single point of failure elsewhere in the topology.

This observation has a direct and, from the perspective of this thesis, more consequential counterpart when the "problem" at the chokepoint is not a fault but a defensive action. A SOAR platform reasoning purely at the level of "block the source of this alert" has no visibility into how many services depend, transitively, on the path that passes through the node it is about to isolate. Chapter 6 formalizes this observation as an explicit derivation rule, and Chapter 7 shows a concrete instance of it in the case study, where a single simulated crash of `fw_int` is shown to independently break three logical connections that a naive, single-hop dependency check would not have connected to one another.

Chapter 6

Threat Model

This chapter presents the formal threat model that generalizes the analysis of Chapter 5, and that the derivation engine of Chapter 7 operationalizes. The model does not stop at a simple and static evaluation of network topology: it integrates the concepts of active defense, the SOAR, the SOC, and its playbooks, together with the dependencies between services, and it evaluates, through the logical exploration of system states, both how a threat can propagate across logical connections or physical adjacencies, and how the countermeasures meant to stop it can, in some specific edge cases, cause disruption and resource saturation of their own (Alert Fatigue). The model is expressed in first-order logic (FOL) and graph-theoretic terms, with the goal of being precise enough to drive an automated derivation engine (Chapter 7) while remaining tractable (and feasible) enough to avoid the state-explosion problems that a fully general model of network security would, almost certainly, incur.

6.1 Attacker Model and Scope

Before introducing the formal apparatus, it is necessary to state precisely what an attacker is assumed to be capable of within this model, and which architectural simplifications have been adopted, since these choices bound the applicability of every result derived from the model.

6.1.1 Attacker Capabilities

- **Entry point.** The attacker does not materialize inside the network from nothing: the simulation requires at least one entity to be declared, as an axiom, as already compromised. In closed, air-gapped scenarios this entity coincides with an insider or an already-infected device. In exposed scenarios,

the public network ("internet", or more generally the external network) is modeled as a macro-node that is permanently compromised and under the attacker's control.

- **Physical constraints.** The attacker's domain is strictly limited to the logical and software domain. The model assumes it is impossible for the attacker to alter the physical topology of the network, for instance by cutting cables, inserting undocumented hardware devices, or physically manipulating router ports. The structure of physical links is therefore immutable from the attacker's perspective.

- **Abstraction of reconnaissance.** To keep the logical engine tractable, the model does not represent a reconnaissance phase (such as port scanning) as a distinct action. It is assumed, implicitly, that whenever the attacker has a communication path toward a target node, it automatically knows which vulnerabilities are present and is capable of exploiting them.

6.1.2 Modeling Limitations and Design Choices

- **Deterministic exploitation.** Vulnerabilities and the outcome of an exploit are evaluated in binary logic (success or absolute failure). The model does not represent success probabilities, CVSS scores, or probabilistic outcomes: if a vulnerability is present and a network vector to it exists, the attack succeeds. This is just a simplification made for keep an easy tractability, in factm real exploitation is of course probabilistic, and modeling it that way is a possible natural direction for future work.

- **Temporal abstraction.** The engine evaluates the logical causality of events, that is, state transitions in a decision tree, rather than chronological metrics. It does not estimate how many seconds a piece of malware takes to propagate; instead, it verifies whether the preconditions for isolating a threat are satisfied before its propagation, in a purely causal sense.

- **Single physical link (no multigraphs).** To prevent the algorithmic complexity that would result from computing paths over multigraphs, the model assumes at most one direct physical link between any two nodes.

6.2 Notation and System Sets

The model uses standard first-order logic connectives and quantifiers ($\forall$, $\exists$, $\in$, $\wedge$, $\vee$, $\neg$, $\implies$), together with two additional symbols that are specific to this framework:

- $\rightarrow$ denotes a *derivation rule*: $A \rightarrow B$ reads "if premise A is satisfied, the engine derives the new state B", as opposed to $\Longrightarrow$, which denotes ordinary logical implication between two conditions that both hold within the same state.

- $:=$ denotes a *structural definition*: $A := B$ reads "concept A is defined by expression B".

Table 6.1 summarizes the full notation, including the two framework-specific symbols above, for reference throughout the rest of this chapter.

Symbol	Logical Meaning	Reading Example
$\forall$	for all	$\forall x, P(x)$: "for every x, $P(x)$ holds"
$\exists$	there exists	$\exists x, P(x)$: "there is some x for which $P(x)$ holds"
$\in$	belongs to	$x \in E$: "x is an entity"
$\wedge$	and	$P \wedge Q$: "P and Q both hold"
$\vee$	or	$P \vee Q$: "at least one of P, Q holds"
$\neg$	not	$\neg P$: "P does not hold"
$\Longrightarrow$	implies	$P \Longrightarrow Q$: "whenever P holds, Q holds too"
$\rightarrow$	derivation rule	$A \rightarrow B$: "if A is satisfied, the engine derives B"
$:=$	structural definition	$A := B$: "A is defined as B"
α	assumed property	$\alpha\, Vulnerable(A, T)$: "declared as input"
κ	derived property	$\kappa\, Compromised(A, T)$: "computed by the engine"

Table 6.1: Notation used throughout this chapter.

To formalize the domain of analysis rigorously, the model is organized within five fundamental sets, which comprehends every variable that appears in the formulas of the rest of this chapter.

- E (**Entities**): the set of all active and passive components. *Example: a DNS server, a firewall and also the SOAR itself.*

- R (**Relations**): the set of structural or logical links between two or more entities. *Example: a web service's dependency on its database, or the physical cable between a switch and a host.*

- T (**Threats**): the set of offensive actions. *Example: a denial-of-service attack, or a remote-code-execution exploit.*
- P (**Properties**): the set of operational, security, or behavioral states assumed by entities. *Example: the state of being vulnerable, compromised, or malfunctioning.*
- PB (**Playbooks**): the set of automated incident-response procedures. *Example: a playbook that instructs a firewall to block a given source.*

The model distinguishes two categories of properties, following the same assumed/derived convention adopted by TAMELESS [12] (Section 3.4):

- **Assumed properties** (α), which represent the primary knowledge of the system's components and must be declared as input to the engine. They are certain facts about the system: the topology, the roles of its components, and the intrinsic vulnerabilities known before the attack begins. As a convention, nominal positive states (e.g., "the SOAR is operational") are never declared explicitly, to avoid redundancy: only the negative exception (e.g., *Offline*) is declared, and the absence of a negative declaration implies correct functioning.
- **Derived properties** (κ), which denote security properties that can or could become true as a consequence of a threat or of a defensive rule firing. They are dynamic, computed at runtime by the inference engine, and allow the exploration of alternative attack paths without corrupting the initial state.

6.3 Network Axioms and Structural Derivations

The rules that describe how an offensive action propagates depend closely on the existence of valid paths through the network. For this reason, this section separates the purely architectural properties, the ones that describe connectivity and routing independently of any attacker, from the security-specific properties introduced in Section 6.4. The predicates below are presented in their neutral, structural form, before any threat is layered on top of them.

Physical link. $Physical_link(A, B)$ denotes the existence of a physical connection between A and B. The relation is symmetric: declaring a link in one direction implies the existence of the reverse link.

Physical Link

$$\alpha\, Physical_link(A, B) \implies \alpha\, Physical_link(B, A)$$

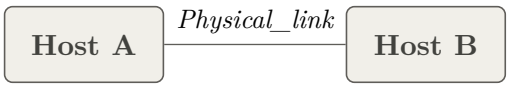


Figure 6.1: A physical link: symmetric, declared once, holds in both directions.

Whitelist. $Whitelist(A, S)$ is a local rule on node A that explicitly authorizes traffic originating from S. A node operates in *default-deny* if at least one whitelist rule is configured on it, otherwise an empty node operates in *default-allow*:

Whitelist

$$\alpha\, Whitelist(fw_int,\ dns)$$



Figure 6.2: Whitelist: node fw_int locally authorizes traffic declared as originating from dns.

Has-Whitelist. An entity with no configured rule falls back to default-allow:

Has Whitelist

$$Has_Whitelist(A) := \exists S \in E,\ \alpha\, Whitelist(A, S)$$



Figure 6.3: Has_Whitelist: a node with no configured rule falls back to default-allow.

Two points of view. Two distinct perspectives recur throughout this model, and it is worth naming them before the first predicate that depends on the distinction. An attacker reasons with every exploit available: a compromised node is, from the attacker's side, simply another way to move traffic, no different in principle from an authorized path. A defender's own policy, by contrast, has no notion of a secretly compromised node: for routing to be considered legitimate from the defense's perspective, it has to work without any exploit at all, evaluated purely against the rules that were actually declared. *Accepts*, below, is the attacker's view; *PolicyAccepts*, introduced immediately after it, is the defender's. Keeping them as two separate predicates, rather than one parameterized by a trust assumption, is what later makes it possible to ask a question that matters in Chapter 7: whether a defensive action actually contains an attacker who already controls the very node

the action targets, rather than silently assuming that blacklisting a source revokes the control it grants.

Down (crash simulation). $Down(A)$ is not a property the engine derives on its own: it is an axiom injected into a hypothetical exploration state to ask what happens if A is simulated as crashed, used specifically by the crash-simulation queries of Chapter 7. It takes priority over every other consideration in *Accepts* below: once simulated as down, A accepts no traffic at all, not even from an attacker who already controls it.

Accepts. $Accepts(A, S)$ is the central routing predicate, and it is the attacker's view. Node A forwards traffic from S under two disjoint conditions: either A has been compromised by the attacker with full control, bypassing network restrictions entirely; or A evaluates its local rules (whitelist, or default-allow in its absence) and is not an actuator to which the SOAR has imposed a global blacklist against S:

Accepts

$$\begin{aligned} Accepts(A,S) := \neg\,\alpha\,Down(A)\ \wedge\ \Big[& \\ & \exists T_A \Big[\kappa\,Compromised(A,T_A) \wedge \alpha\,Impact(T_A, Control)\Big]\ \vee \\ & \Big[\big(\neg Has_Whitelist(A) \vee \alpha\,Whitelist(A,S)\big)\ \wedge \\ & \ \neg\big(\alpha\,Actuator(A) \wedge \kappa\,Blacklisted(S)\big)\Big]\Big] \end{aligned}$$

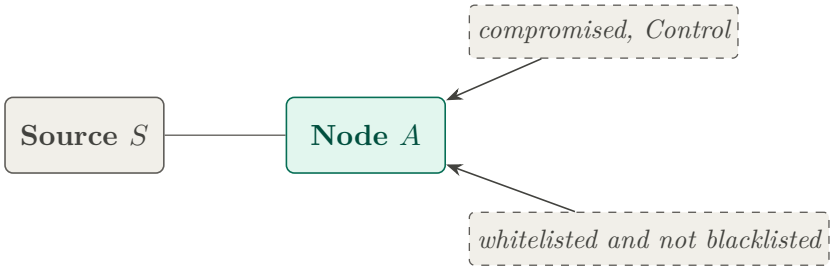


Figure 6.4: Accepts: a node forwards traffic from S either by being bypassed through compromise, or through ordinary, unblocked routing.

PolicyAccepts. The defender's view, evaluating only local policy and dropping the compromise-bypass disjunct entirely, is what this thesis calls the purely defensive view:

PolicyAccepts

$$\begin{aligned} PolicyAccepts(A,S) := &\big(\neg Has_Whitelist(A) \vee \alpha\, Whitelist(A,S)\big) \\ &\wedge\ \neg\big(\alpha\, Actuator(A) \wedge \kappa\, Blacklisted(S)\big) \end{aligned}$$

Path. A valid, loop-free, unidirectional network trace, in which every intermediate node accepts traffic originating from the source:

Path

$$\begin{aligned} \kappa\, Path(S,D) := \exists(n_1,\ldots,n_k) \text{ s.t. } \ & n_1 = S \wedge n_k = D \\ & \wedge\ \forall i < k,\ \alpha\, Physical_link(n_i, n_{i+1}) \\ & \wedge\ \forall i \neq j,\ n_i \neq n_j \\ & \wedge\ \forall i \geq 2,\ Accepts(n_i, S) \end{aligned}$$



Figure 6.5: A Path: a loop-free chain of physical links in which every intermediate node accepts traffic from S.

Logical connection. An abstraction of a complex service interaction (e.g. an API call, a web portal request), requiring the structural existence of two valid unidirectional paths, one in each direction:

Logical Connection

$$\kappa\, Logical_connection(A,B) := \kappa\, Path(A,B) \wedge \kappa\, Path(B,A)$$

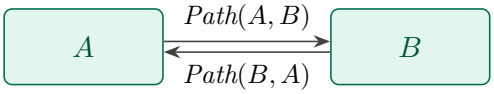


Figure 6.6: Logical_connection: two valid unidirectional paths, one in each direction.

Depends-on. $Depends_on(A,B)$ expresses that the operability of A is subordinate to B. Parameters can be either entities or relations (logical connections); a

cascading approach (Entity → Relation → Entity) is preferred over a direct Entity-to-Entity link, which is reserved for dynamics external to the network context (e.g. shared power supply):

Dependency Cascade (Example)

$$\alpha\, Depends_on(Web\ App,\ Web\ Server)$$
$$\alpha\, Depends_on(Web\ App,\ \kappa\, Logical_connection(Web\ App, DB))$$
$$\alpha\, Depends_on(\kappa\, Logical_connection(Web\ App, DB),\ DB)$$



Figure 6.7: Depends_on connection of [Entity → Relation → Entity].

6.4 Security State and Defense Predicates

This section formalizes the dynamics of compromise, the propagation of service disruption, and the automated reactions of the infrastructure, distinguishing rigorously between the different types of impact an entity can suffer.

Vulnerable and Impact. $Vulnerable(A, T)$ denotes that entity A has an intrinsic weakness exploitable by threat T. $Impact(T, Type)$ classifies the architectural impact of a threat into one of four categories: *Availability* (service disruption, for example DoS or ransomware), *Confidentiality* (data theft or exfiltration), *Integrity* (tampering with data or responses), and *Control* (the attacker gains executive privileges on the node, e.g. through RCE). Only the *Control* impact allows the attacker to reuse the node for lateral movement. A single threat can be declared with more than one impact, leaving the engine free to choose which one to exploit depending on the branch of the exploration.

Vulnerable and Impact (Example)

$$\alpha\, Vulnerable(db,\ cve_db_exfil)\ \wedge\ \alpha\, Impact(cve_db_exfil,\ Confidentiality)$$

Compromised. $Compromised(A, T)$ denotes that entity A has suffered a successful violation by threat T. At the logical level, this state always presupposes a base condition: the entity must be vulnerable to the threat, $Vulnerable(A, T)$.

When provided as an axiom at configuration time, this predicate defines the attacker's original entry point; for formal consistency, declaring a node as initially compromised implies that it possesses the corresponding vulnerability.

Compromised (Example)

$$\alpha\, Compromised(internet,\ initial_access)$$

Malfunction. $Malfunction(A)$ denotes that entity A is in a state of fault or service disruption. When provided as an axiom, it represents a device already out of service at the start of the simulation (e.g. hardware failure or maintenance), independent of any cyber vulnerability. When derived (κ), it presupposes that the entity has been compromised by a threat with an impact specifically on availability, *or*, as detailed in Section 6.5, that it is unreachable as a consequence of the defensive posture of the network itself.

Malfunction (Example)

$$\alpha\, Malfunction(fw_int)$$

Sensor and Actuator. $Sensor(A)$ denotes a monitoring device: it generates alerts if and only if it is crossed by the trace of a threat, and is able to deliver them if and only if it possesses an active logical connection toward the orchestrator. $Actuator(A)$ denotes an enforcement device (e.g. a firewall or a Network Access Control system): it receives the global blacklist and blocks the forwarding of packets from the banned source, acting dynamically on the *Accepts* predicate.

Sensor and Actuator (Example)

$$\alpha\, Sensor(ids)\ \wedge\ \alpha\, Actuator(fw_p)$$

Offline and Busy. $Offline(A)$ and $Busy(A)$ force an initial disruption, identifying respectively a technical bottleneck (e.g. the SOAR under maintenance) and a human one (e.g. a SOC analyst engaged in other operations). Both are keyed to the same orchestrator identifier in the engine: there is no separate entity for the analyst, only a second dimension of unavailability attached to the orchestrator itself.

Offline and Busy (Example)

$$\alpha\, Offline(soar)\ \wedge\ \alpha\, Busy(soar)$$

Macro-states. To prevent ambiguity between disruption and compromise (an infected but functioning system versus a clean but inaccessible one), three architectural macro-states are defined:

Macro-States

$$\kappa\,Operational_State(A) := \neg\kappa\,Malfunction(A)$$
$$\kappa\,Safe_State(A) := \neg\exists T,\ \kappa\,Compromised(A,T)$$
$$\kappa\,Valid(A) := \kappa\,Operational_State(A) \wedge \kappa\,Safe_State(A)$$

The first state denotes mere operability; the second denotes the absence of a successful exploit; the third, *Valid*, denotes an entity that is simultaneously operational and secure, the condition a component would need to satisfy to be considered fully trustworthy by the rest of the model.

6.5 Derivation Rules

These rules process the axioms and the computed paths of the previous two sections to expand the search tree, simulating the causal chains of attack and defense.

6.5.1 Malfunction and Availability Loss

The state of disruption, $\kappa\,Malfunction$, is the most structurally rich derived property in the model, because it can be triggered by four distinct, independent causes. A compromise with an availability impact does not, on its own, automatically cause a malfunction: it makes the entity a *candidate* for the crash-simulation queries of Chapter 7, which inject *Down* explicitly, in a separate hypothetical state, rather than have it follow automatically from compromise. The first two describe ways a malfunction can originate at the entity itself, whether from an inherent fault or from a hypothetical crash injected for exploration; the remaining two describe how a malfunction originating elsewhere propagates through the dependency graph, formalizing the amplification pattern Chapter 5 introduced, and that Section 5.4 showed to be far from marginal in practice.

Malfunction and Availability Loss

$$\alpha\, Malfunction(A) \rightarrow \kappa\, Malfunction(A) \qquad \text{(native fault)}$$

$$\alpha\, Down(A) \rightarrow \kappa\, Malfunction(A) \qquad \text{(simulated crash)}$$

$$\alpha\, Depends_on(A, B) \land B \in E \land \kappa\, Malfunction(B) \rightarrow \kappa\, Malfunction(A) \qquad \text{(entity cascade)}$$

$$\alpha\, Depends_on(A, \kappa\, Logical_connection(A, B)) \land \Big[\neg\kappa\, PolicyLogical_connection(A, B) \lor\ \kappa\, Malfunction(B)\Big] \rightarrow \kappa\, Malfunction(A) \qquad \text{(connection cascade)}$$

The first two rules reach the same conclusion, $\kappa\, Malfunction(A)$, from different starting points, and in particular, the difference between them lies in where they come from: once $\kappa\, Malfunction(A)$ holds, the cascade rules that follow treat a native fault and a simulated crash identically. *Native fault* declares a fact about the scenario itself: this specific node is already broken when the simulation begins, once, as part of the case study's configuration. *Simulated crash*, by contrast, is not a fact about the scenario at all: it is how the crash-simulation queries of Chapter 7 ask, for any node in turn, what would happen if it went down, without editing the underlying configuration to do so. This second point is not only conceptual, it is also what keeps the crash-simulation mechanism safe to use. Reusing $Malfunction$ for both roles would require each query to assert the fact for the node under test, evaluate the network under it, and then retract it before moving on to the next candidate; forgetting that last step is exactly the kind of mistake that produces no error and a wrong answer, since every query run afterward in the same session would silently keep treating the node as broken. Keeping *Down* as a separate predicate, threaded through the hypothetical state of a single query rather than written into the shared configuration, makes that hypothetical nature structural: it cannot outlive the query that introduced it, because it was never asserted as a persistent fact to begin with.

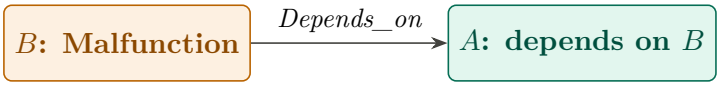


Figure 6.8: Malfunction propagation (entity cascade): a fault in B (orange, a state of disruption) is inherited by A through the dependency alone, not through any compromise of A.

The fourth rule, *connection cascade*, is the formal counterpart of the amplification pattern discussed in Section 5.3. It checks the connection against the policy-only view of Section 6.3, rather than the unrestricted one: it asks whether the connection holds up under ordinary policy, independent of whether a compromised endpoint

could still force it open. In the engine, this single rule corresponds to three separate clauses, one for the remote endpoint independently malfunctioning, and two more, structurally identical, for a bidirectional or a unidirectional connection broken by policy respectively; they are collapsed into one rule here because they share the same premise and consequence. It fires under two structurally different circumstances, both captured by the same rule: either the logical connection A depends on can no longer be established at all, because a chokepoint on its underlying physical path has become unreachable (regardless of whether the remote endpoint B is healthy); or the remote endpoint B has itself independently entered a malfunction state. Chapter 7 shows that, in the case study, it is specifically the first branch of this rule, and not a chain of compromise, that accounts for the multi-node availability loss produced by the failure of a single chokepoint node.

Optional: A Faster Quarantine Shortcut (disabled by default)

A node placed on the global blacklist can, optionally, be treated as itself malfunctioning outright, rather than only through the connection-cascade rule above:

$$\kappa\, Blacklisted(A) \rightarrow \kappa\, Malfunction(A) \qquad \text{(quarantine)}$$

This is sound only under the same precondition noted in Section 5.4, that every path a dependent entity relies on passes through an actuator; it is easy to state but not, in general, correct, and connection cascade already covers the general case through *PolicyLogical_connection*, which consults *Blacklisted* on its own. For this reason the engine's default behavior is the fully general one: the quarantine derivation (lines 427–428 of the implementation) and its matching diagnostic clause in the backward-search (lines 1245–1250) are present in the source but commented out, to be re-enabled only once a given deployment is known to satisfy the precondition above. Disabling them was validated against the full query battery for this case study: only entities with no *Depends_on* relation of their own, such as `fw_p` or `host1`, stop registering as malfunctioning purely from being blacklisted, and every other outcome is unchanged.

6.5.2 Compromise and Lateral Movement

The attacker can pivot from a compromised node B to a node A if two conditions hold jointly: B has been compromised by a threat that grants full control ($\alpha\, Impact(T_B, Control)$), since threats that only target availability or confidentiality do not provide the executive access necessary for the jump, and a usable vector

exists, either an application-level logical connection toward A, or a simple physical adjacency enabling a low-level attack:

Compromise and Pivoting

$$\alpha\, Compromised(A, T) \rightarrow \kappa\, Compromised(A, T)$$

$$\kappa\, Compromised(B, T_B) \wedge \alpha\, Impact(T_B, Control) \wedge$$

$$\Big[\kappa\, Logical_connection(B, A) \vee \alpha\, Physical_link(B, A)\Big] \wedge\ \alpha\, Vulnerable(A, T_A) \rightarrow \kappa\, Compromised(A, T_A)$$

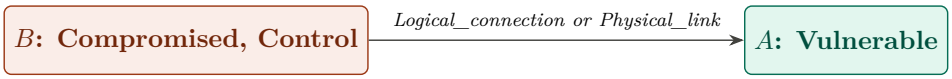


Figure 6.9: Pivoting: full control over B, plus a usable vector, is enough to compromise A.

6.5.3 Detection and Alert Fatigue

A sensor detects a trace C (the physical path underlying a logical connection) that is active from a compromised node. To alert the central system, it must possess an active logical connection to the orchestrator. A sensor that has itself been compromised with full control, however, stops counting as an independent witness on that same trace: it is no longer reporting on the attacker, it is possessed by the attacker, and the engine excludes exactly this one impact type from the sensor's own compromise, not compromise in general, since a confidentiality leak on a sensor does not by itself make its reports untrustworthy. If the attacker gains *Control* over the sensor itself, it may additionally generate fabricated reports to induce logical and human saturation (*Alert Fatigue*):

Detection and Alert Fatigue

$$\Big(\kappa\, Compromised(S, T_S) \vee \kappa\, Compromised(D, T)\Big) \wedge S \in C \wedge D \in C \rightarrow \kappa\, Active_Threat(C, T)$$

$$\alpha\, Sensor(Sens) \wedge Sens \in Nodes(C) \wedge$$
$$\neg\, \exists T_{Sens},\ \kappa\, Compromised(Sens, T_{Sens}) \wedge \alpha\, Impact(T_{Sens}, Control) \wedge$$
$$\kappa\, Active_Threat(C, T) \wedge \kappa\, Logical_connection(Sens, SOAR) \rightarrow \kappa\, Alert(C, T, Sens)$$

$$\alpha\, Sensor(Sens) \wedge \kappa\, Compromised(Sens, T_{Sens}) \wedge \alpha\, Impact(T_{Sens}, Control)$$
$$\wedge\ \kappa\, Logical_connection(Sens, SOAR) \rightarrow \kappa\, Alert(\emptyset, AlertFatigue, Sens)$$

This is a different failure mode from the one Section 6.6 formalizes. Here, a compromised sensor on a genuine trace goes silent: the attack still happens, but

the one witness who would have reported it no longer counts, and nothing appears to have been reported at all. There, a compromised sensor actively reports on an entity the attack never touched. The first is the defense failing to notice; the second is the defense being turned against a target of the attacker's choosing. Both share the same precondition, control over the sensor, but they are not the same event, and a model that captured only one of them would miss half of what a hijacked sensor can actually do.

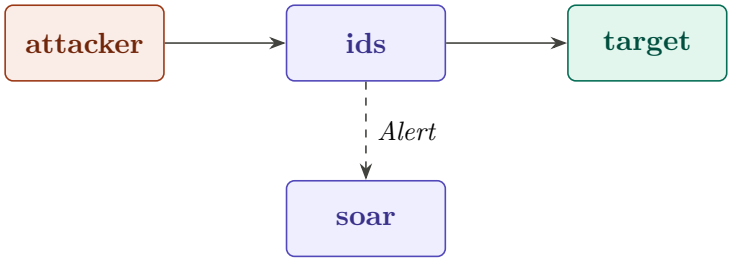


Figure 6.10: Detection: the sensor sits on the attacker's trace, and, independently, delivers an alert to the SOAR only if its own logical connection to the orchestrator is up.

Indirect detection through a hidden chokepoint. A sensor does not need to appear as an explicit node of an attack path to detect it. When a step of the path crosses a logical connection between two entities *Src* and *Dest*, the physical route that connection actually uses is hidden by the abstraction: the path lists *Src* and *Dest*, not the firewalls and sensors the traffic passes through to get from one to the other. A sensor positioned on that hidden route can still be an unavoidable waypoint for it, even though it is never named in the path. The engine checks this the same way it checks a simulated crash: by asking whether the connection between *Src* and *Dest* would still hold if the sensor were removed.

Indirect Detection via a Hidden Chokepoint

$$\kappa\, Chokepoint(Sens, Src, Dest) := \kappa\, Path(Src, Dest) \wedge \neg\, \kappa\, Path(Src, Dest)\big|_{\alpha\, Down(Sens)}$$

$$\alpha\, Sensor(Sens) \wedge \exists\, Src, Dest \in C,\ \kappa\, Chokepoint(Sens, Src, Dest) \wedge \kappa\, Logical_connection(Sens, SOAR) \rightarrow \kappa\, Alert(C, T, Sens)$$

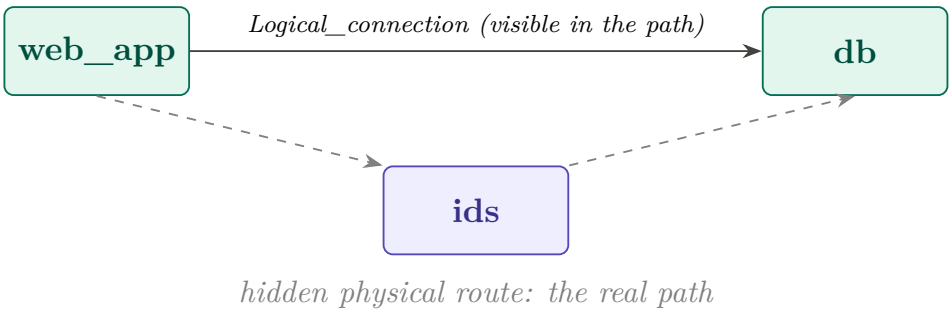


Figure 6.11: Indirect detection: `ids` never appears in the attack path shown to the analyst, but removing it would break the logical connection the path relies on, so it still detects the traffic passing through it.

6.5.4 Active Defense and Global Blacklisting

The SOAR analyzes generated alerts through playbooks, modeled as a conditional logical tuple $PB = \langle \tau, C, A \rangle$. Each component defines a precise phase of the automated reaction:

- τ (**Trigger**): the triggering event. A playbook fires if and only if $Match(Alert, \tau)$ holds, that is, the incoming alert matches the specific case that playbook is built to handle.
- C (**Conditions**): a list of condition-types, all of which must hold for the playbook to fire. Two conditions, if present in the playbook, are checked: *online*, requiring that the SOAR is not $\kappa\, Offline$, and $human_free$, requiring that the SOC analyst is not $\kappa\, Busy$.
- A (**Actions**): a list of actions, extracted through $Actions(PB)$, each triggering its own effect once the playbook executes. $Block(S)$ derives $\kappa\, Blacklisted(S)$; $Offline(SOAR)$ derives $\kappa\, Offline(SOAR)$; $Busy(SOC)$ derives $\kappa\, Busy(SOC)$.

Crucially, C and A are independent lists, in particular, there isn't a direct implication which imply that in a specific playbook a $human_free$ request must put the SOC Analyst in busy state. A quick approval and an hours-long investigation can both be gated behind the same condition and still have entirely different actions. Whether a given execution occupies the analyst is determined solely by whether $Busy(SOC)$ is itself one of that playbook's declared actions.

Active Defense and Global Blacklisting

$$\begin{aligned} \kappa\, Executed(PB, Alert) := {} & Match(Alert, \tau_{PB}) \\ & \wedge \ \big[online \in Conditions(PB) \implies \neg\kappa\, Offline(SOAR)\big] \\ & \wedge \ \big[human_free \in Conditions(PB) \implies \neg\kappa\, Busy(SOC)\big] \end{aligned}$$

Once $\kappa\, Executed(PB, Alert)$ holds, each action in $Actions(PB)$ triggers its own consequence:

$$\begin{aligned} Block(S) \in Actions(PB) &\to \kappa\, Blacklisted(S) \\ Offline(SOAR) \in Actions(PB) &\to \kappa\, Offline(SOAR) \\ Busy(SOC) \in Actions(PB) &\to \kappa\, Busy(SOC) \end{aligned}$$

The independence between C and A is easiest to see instantiated: two playbooks can share a condition and still differ entirely in what executing them does.

Playbook	Trigger	Conditions	Actions
$pb_isolate_threat$	$malicious_traffic$	*soar* online, analyst free	$Block(S)$
$alert_fatigue$	$alert_fatigue$	*soar* online	$Offline(soar)$
$pb_incident_response$	$ransomware_detected$	*soar* online, analyst free	$Busy(soar)$

Two of the three require $human_free$, yet only one lists $Busy(soar)$ among its actions. These are the exact playbooks the case study of Chapter 7 validates against.

Since $\kappa\, Executed$ is itself conditioned on the state before it, and disabling the SOAR or the analyst can only ever remove options along a branch of exploration, the defensive process can collapse under its own noise without any single step behaving unexpectedly: a playbook triggered by that noise is processed like any other alert, and once the SOAR it depends on is itself $\kappa\, Offline$, every playbook that needed it online is disabled along that same branch, including the one that caused it.

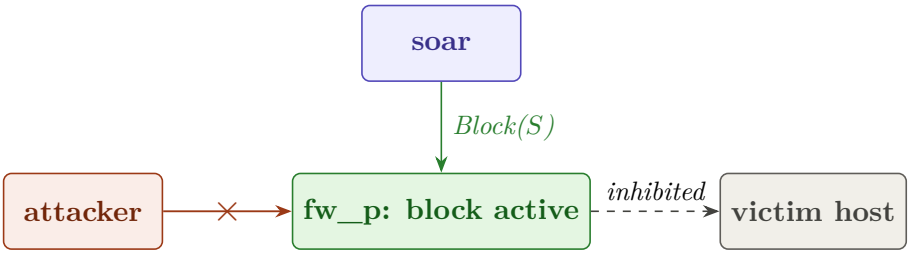


Figure 6.12: Reaction: the SOAR updates the blacklist, the actuator blocks the attacker's traffic (marked), and any traffic further downstream toward the protected host remains inhibited as well.

Note: Extending Beyond the Orchestrator

$Offline$ and $Busy$ are checked and derived here only for $SOAR$ and SOC, the only orchestrator this case study declares. Nothing about the mechanism itself requires this: a future extension of the engine could check or drive the same two condition types, or further ones, against other nodes.

6.6 Modeling SOAR Manipulation

The rules of Section 6.5.3 already allow a hijacked sensor to saturate the SOAR with noise. A hijacked sensor, however, can do something considerably more targeted than generating noise: because the SOAR has no independent way to verify that an alert genuinely corresponds to the node it names, a sensor under the attacker's control can *fabricate* an alert that blames an arbitrary, uncompromised node, and thereby redirect an automated containment action against a target that was never part of the attack. This mechanism is the most direct realization of the risk described in Chapter 1, adversarial alert injection turning the defense into an

instrument of the attack, and it is formalized here as an explicit extension of the model.

Fabricated alert. An entity $Target$ that has never been compromised can nonetheless become the subject of an alert, provided a sensor under the attacker's control is positioned to reach the orchestrator, and a playbook exists whose trigger matches an alert naming $Target$:

Fabricated Alert

$$\begin{aligned} &\alpha\, Entity(Target) \wedge \neg\exists T,\ \kappa\, Compromised(Target, T) \wedge \\ &\exists Sens,\ \Big[\alpha\, Sensor(Sens) \wedge \kappa\, Compromised(Sens, T_{Sens}) \wedge \\ &\alpha\, Impact(T_{Sens}, Control) \wedge \kappa\, Logical_connection(Sens, SOAR)\Big] \wedge \\ &\exists PB,\ \tau,\ Match\big(alert(AlertType, Target), \tau\big) \rightarrow \\ &\qquad \kappa\, AlertFabricated(Target, AlertType) \end{aligned}$$

Action possible. A defensive action becomes derivable through a genuine alert (Section 6.5.4) or through a fabricated one, provided the playbook's conditions are otherwise satisfied:

Action Possible

$$\begin{aligned} &\kappa\, AlertFabricated(Target, AlertType) \wedge \exists PB,\ Match\big(alert(AlertType, Target), \tau_{PB}\big) \\ &\wedge\ C_{PB} \text{ satisfied} \wedge\ \ Action \in Actions(PB) \rightarrow \kappa\, ActionPossible(Action) \end{aligned}$$

Together with the *connection cascade* rule of Section 6.5.1, this means that an uncompromised node can lose availability purely as a consequence of a fabricated alert against a *different* node it structurally depends on, a chain that involves no genuine compromise at any point.

Playbook neutralization. Because every playbook's conditions require the SOAR to be operational, and playbooks that require human approval additionally require the analyst to be free, a defensive action that drives the SOAR $Offline$ or the analyst $Busy$ can disable playbooks other than the one that produced it.

This is not paradoxical when PB is the playbook whose action produced $Offline(SOAR)$: once the SOAR is offline, every playbook that requires it to be online, itself included, is disabled along that branch of exploration, exactly as the rule predicts.

Chapter 7

Implementation and Validation

This chapter describes how the threat model of Chapter 6 was operationalized into a working derivation engine, and reports the validation performed on the case study topology of Section 5.4. The discussion focuses on architectural choices and on the outcomes of the validation process; source code itself is not reproduced here.

7.1 Implementation Overview

The derivation engine was implemented in Prolog. This is a decision made naturally from the structure of the model itself. The model is, in other words, expressed as a set of axioms and derivation rules over a relational domain, and the native resolution mechanism of Prolog that includes unification and backtracking search over a database of facts and rules, later maps onto this structure with little translation overhead. In particular, the exploratory nature of the model, where multiple alternative attack paths, sensor placements, or in general defensive outcomes must all be considered as independent branches and corresponds naturally to the built-in backtracking present in Prolog, without requiring a specific search algorithm to be written.

7.2 Validation Methodology

The engine was validated on the case study topology of Section 5.4 using a battery of 22 queries, covering every query type introduced in Section 7.3 and deliberately probing several edge cases: an integrity check of the knowledge base itself; the full attack surface of every asset in the topology, including the two actuators

and the sensor itself (a sensor targeted directly tests whether the reliability rule of Section 6.5.3 applies to a compromise of a sensor); seven of the backward-search queries, two of which are specifically regression checks on nodes with no dependencies, to confirm the removal of the quarantine shortcut of Chapter 6 did not introduce a false malfunction; comparisons of the reachable surface before and after a defensive action in which both for a single entry point and for both automatically at once; the residual attack surface after a chokepoint crash, from both the entry points, and last but not least, the set of playbooks that can be disabled by manipulating the specific state of the SOAR.

Rather than treating the engine as a black box whose output is accepted at face value, each result was checked through a combination of three complementary methods.

1. **Manual trace verification.** For a representative subset of results, in particular every reported attack path and every detection outcome, the derivation was reconstructed by hand against the axioms of Chapter 6, to confirm that the reported outcome, and specifically the reported *absence* of an outcome, followed from the stated rules rather than from an implementation artifact.

2. **Independent review by multiple different readers.** The engine and its own output were also reviewed independently by several different readers, both human and AI-based (Claude Sonnet 5 High, Gemini Pro, and ChatGPT Pro were used for the task), each were asked to propose some edge cases or particular/specific scenarios not yet covered by the existing list of queries, without being told what the other reviewers had already discovered. In every case any reviewer claim was resolved the same way, in other words by manual derivation against the formal rules of Chapter 6, which remains the final arbiter. A reviewer's opinion alone was never taken as confirmation on its own. At the end, several of the regression queries in the battery above exist specifically because a reviewer raised them. In the version reported in this chapter, no reviewer, human or AI, identifies a further discrepancy.

3. **Differential testing across engine revisions.** After a change was applied to the engine, the full battery of queries was re-executed, and its output was compared, line by line, against the output produced before the change. This made it possible to confirm not only that the intended fix took effect, but that no other query was affected, a property that is not guaranteed a priori in a declarative language where predicates can share dependencies in non-obvious ways.

7.3 Case Study Results

This section walks through a representative subset of the 22-query battery, chosen so each one illustrates a distinct mechanism of the model, not the same pattern repeated with different node names. Every trace below is quoted from the engine's actual output, read in the direction the attack really travels: source first, target last, the reverse of how the engine itself prints it.

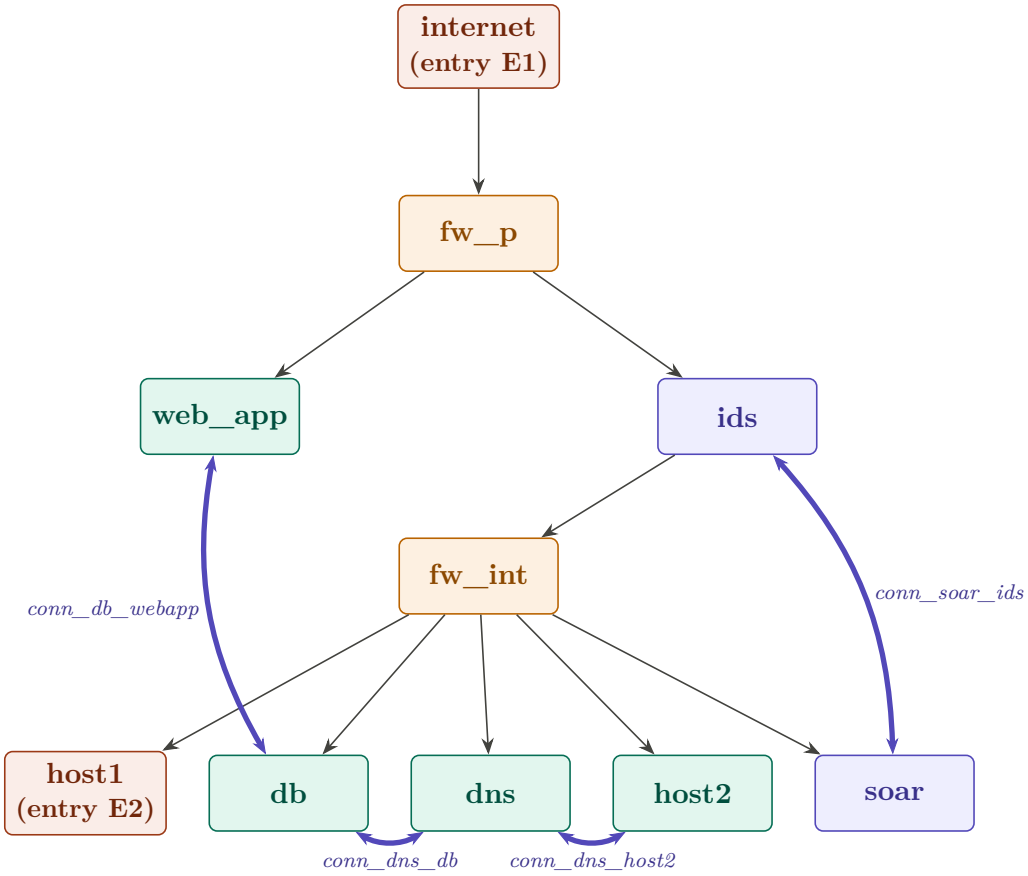


Figure 7.1: The case study topology used for the query battery. Coral: entry points under attacker control. Orange: actuators, the chokepoints of Chapter 5. Purple: sensor and orchestrator. Teal: protected assets. Single arrows are physical links; double arrows are logical connections, drawn bidirectional here since only their existence matters. Every trace in the rest of this section is read against this map.

Table 7.1 collects every declared vulnerability in the case study, in the order the topology of Figure 7.1 is normally read: the two entry points first, then `fw_p`'s side of the network, then `fw_int`'s. Every trace in the rest of this section names one

of these vulnerabilities; this table is the reference to check back against. `fw_int` and `dns` are the only entities with more than one, which is why each of them can support more than one kind of attack independently.

Entity	Vulnerability	Impact
`internet`	`initial_access`	Control
`host1`	`credenziali_rubate`	Control
`fw_p`	`cve_fw_p`	Control
`web_app`	`cve_web_app`	Control
`ids`	`cve_ids`	Control
`fw_int`	`cve_fw_int`	Control
`fw_int`	`cve_fw_int_dos`	Availability
`db`	`cve_db_exfil`	Confidentiality
`dns`	`cve_dns`	Control
`dns`	`dns_spoofing`	Integrity
`host2`	`cve_host2_exfil`	Confidentiality
`soar`	`cve_soar_exfil`	Confidentiality

Table 7.1: Every declared vulnerability in the case study topology, with the impact type it carries (Section 6.4). Only a *Control* impact allows lateral movement under the Pivoting rule of Section 6.5.2.

7.3.1 Attack Surface Highlights

`asset_threat_surface(Target)` reports every way a given asset can be compromised directly, lose availability through a dependency chain, or lose availability through a fabricated alert (Section 6.6) with nothing actually compromised. It was run against every node in the topology, entry points aside; among its results, the two cases below are walked through in full, both against the same target, `db`, because together they show the full range of what the engine reports for a single asset: one route in, and one route out of, the reach of the only sensor in the network.

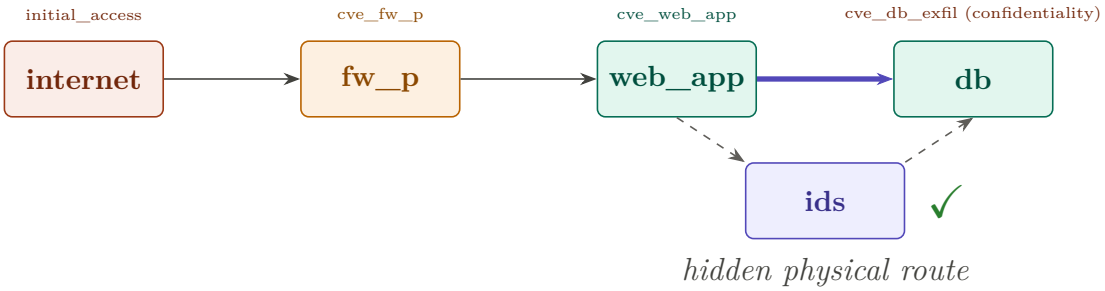


Figure 7.2: Case 1: an external attacker, detected. An attacker enters through `internet`, compromises `web_app`, and reaches `db` over the logical connection `conn_db_webapp` (single arrows are physical links; the double arrow is the logical connection, drawn in the direction this trace uses it; the thin gray dashed lines are not a connection at all, just an illustration of the hidden physical route). The printed path never names `ids`, but it is an unavoidable waypoint for that connection's physical route, so it detects the trace anyway (Section 6.5.3, Indirect Detection via a Hidden Chokepoint). Worst case, the SOAR reacts by blocking both `web_app` and `db`.

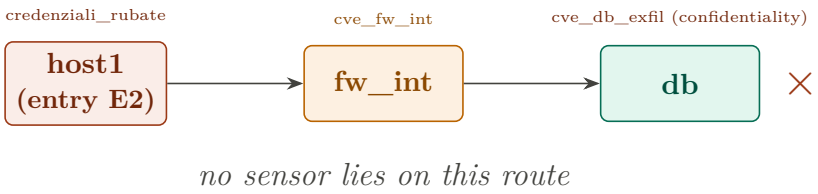


Figure 7.3: Case 2: an insider, undetected. The same asset, `db`, reached by the insider entry point instead: `host1` to `fw_int` to `db`, two physical hops (both solid arrows), no logical connection involved. `ids` sits nowhere on this route, so nothing reports it: `NOT DETECTED`.

Two cases against the same target. `db` has six routes in total across the two entry points; only Case 1 is ever reported as detected, and every other path to every other asset in the battery is `NOT DETECTED` on the same grounds as Case 2: a route either happens to cross the one sensor's hidden physical footprint, or it does not, and `ids` cannot report what never reaches it.

Two further results from the same battery are worth a mention without a diagram of their own. Targeting `ids` directly is the one case that exercises the reliability rule of Section 6.5.3 against the sensor's own compromise, not someone else's: both paths that reach `ids` go undetected, since a sensor compromised with *Control* impact is excluded from counting as a witness on its own trace. Targeting `soar`, in turn, confirms that reachability and pivot capability are separate questions:

`soar` stays reachable from both entry points, but its only vulnerability carries a *confidentiality* impact, never *Control*, so the Pivoting rule of Section 6.5.2 never lets any trace in the battery reuse it as a stepping stone toward another node.

7.3.2 Root-Cause Diagnosis

`run_backward_search(Target, HypotheticalState)` explains why a specific target would, or would not, be affected under a hypothetical addition to the state, such as a node being blocked or simulated as down. It was used, among other things, to confirm that the removal of the quarantine shortcut (Chapter 6) behaves exactly as predicted, on a case where two independent rules agree on the same conclusion rather than contradicting each other.

`run_backward_search(web_app, [block(db)])` triggers two independent, simultaneously correct explanations for the same fact:

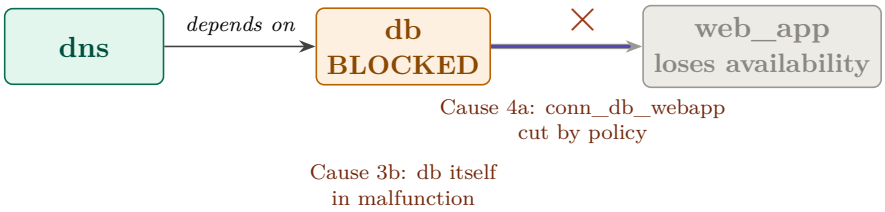


Figure 7.4: Example 1: two causes, one outcome. Blocking `db` makes `web_app` lose availability through both the entity cascade (`db` is itself malfunctioning, shown by its own state) and the connection cascade (`conn_db_webapp` no longer holds under policy, shown by the broken double arrow) at once. Harmless redundancy, not a contradiction: the two rules agree.

7.3.3 Defense Comparison

The query `blast_radius_comparativo(Entry, Threat, DefendedNode)` reports how many node/threat-impact combinations are reachable from an entry point, once without any defense and once with one node blocked. It is deliberately minimal: it does not call `check_integrity` or any other query on its own, so getting a wider picture means chaining those queries by hand, which keeps each call fast.

`blast_radius_comparativo(host1, credenziali_rubate, host1)` finds 10 combinations reachable before the block, and none after it:

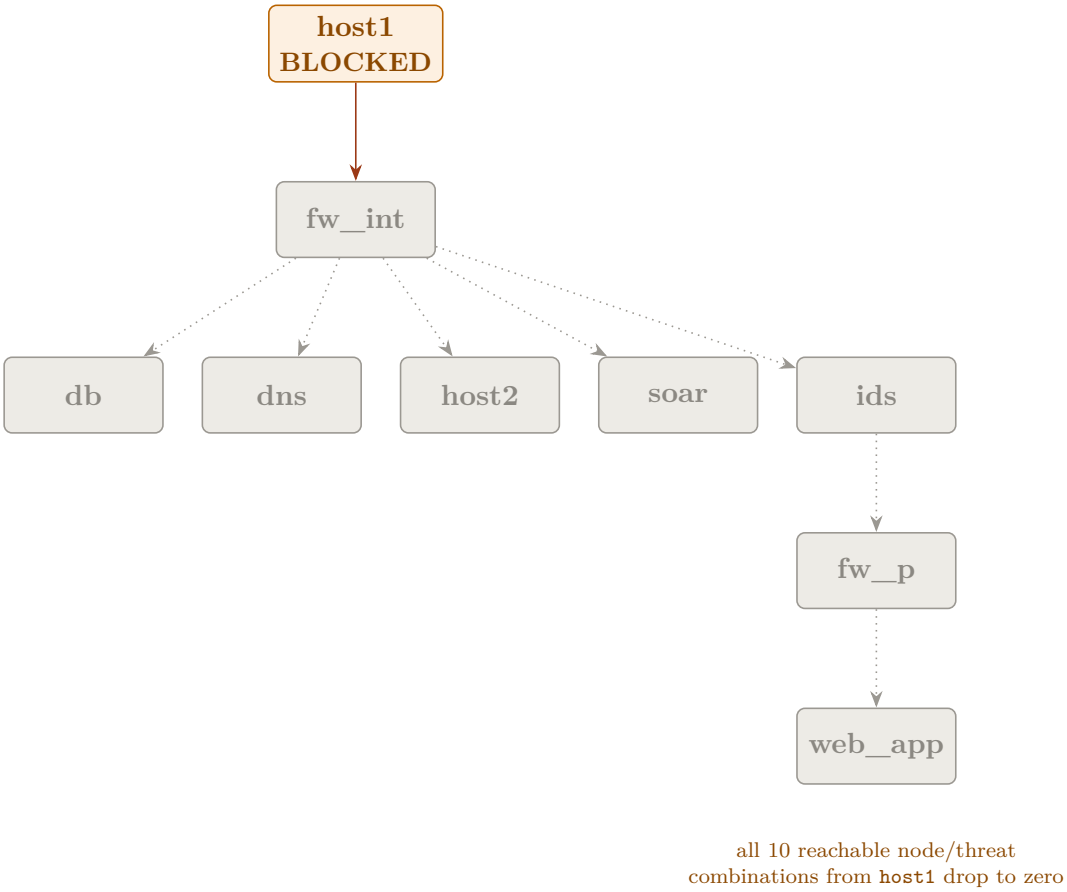


Figure 7.5: Example: quarantining the compromised device. With `block( host1 )` in place, every other node reachable from `host1` becomes unreachable all at once: `fw_int` is `host1`'s only physical neighbor, so quarantining the source itself is enough to break every downstream path in one action.

At this point running this comparison only by hand for every declared entry point does not scale much better than the manual reasoning this thesis is trying to replace, especially once a network has more than the two used in this case study. `blast_radius_tutti_ingressi` exists for exactly that particular reason. In fact, it sweeps every declared entry point automatically, one full report per point so that which breach point causes which consequence is never lost in the aggregation. Run here, it repeats the same comparison for `internet` and `host1` in turn; both drop to zero, for the reason the rest of this section explains.

Section 5.4 already established that neither `internet` nor `host1` is ever whitelisted by the first actuator on its path, a property of this case study's declared configuration and not a structural feature of the model itself, so that reaching anything beyond it depends entirely on compromise, with no explicit authorization involved

at any point. That fact, not just the block, is what the drop from 10 to zero actually reflects: the `[SENZA DIFESE]` count above uses the unrestricted, attacker's view of Section 6.3 (*Accepts*), which lets a compromised entry bypass a policy check it was never authorized to pass; the `[CON DIFESE]` count switches to the pure policy view (*PolicyAccepts*), the same distinction Section 7.3.6 examines directly. Under that second view, the entry was never getting past the first actuator, block or no block. The zero is real, but it is not purely the effect of the block: part of it is the effect of asking a different question.

7.3.4 Crash Simulation: The Chokepoint Revisited

`superficie_crash_disponibilita(Entry, Threat)` answers a question worth asking about any chokepoint independently of whether an attacker is involved at all: if this one node stopped working right now, a hardware fault, a maintenance mistake, anything, what is the actual blast radius? It uses the *Down* mechanism of Section 6.3 to simulate exactly that, not a real compromise, and for a given entry point it reports two disjoint things: which nodes the entry can still reach for compromise once the crash is in place, and, separately, which nodes lose availability purely as a side effect of the crash, through the cascade rules of Section 6.5.1, whether or not the entry could ever have reached them to begin with. It was run against `fw_int`, the chokepoint Chapter 5 argues is structurally overloaded, asking the same question from each entry point in turn, since a chokepoint's blast radius does not have to look the same from every direction.

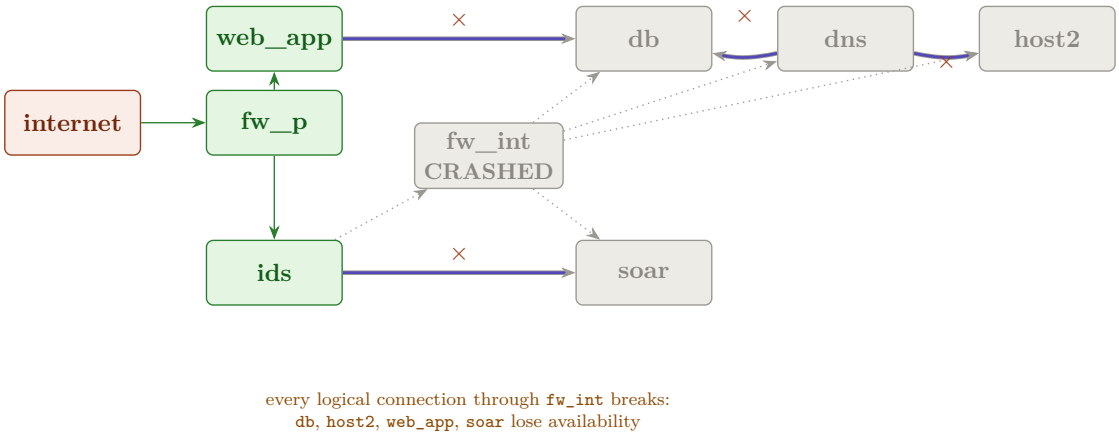


Figure 7.6: Example 1: the external attacker's perspective. Crash of `fw_int` seen from `internet`. `fw_p`, `web_app`, and `ids` sit on the near side of the crashed node and stay reachable: `web_app` connects directly to `fw_p`, never crossing `fw_int` at all. `db`, `dns`, `host2`, and `soar` do not: the only physical neighbor of the `soar` is `fw_int` too, so its logical connection to `ids` breaks along with every other device, cutting the orchestrator off from its own sensor . Four nodes lose availability regardless, through the dependency chains of Chapter 5.

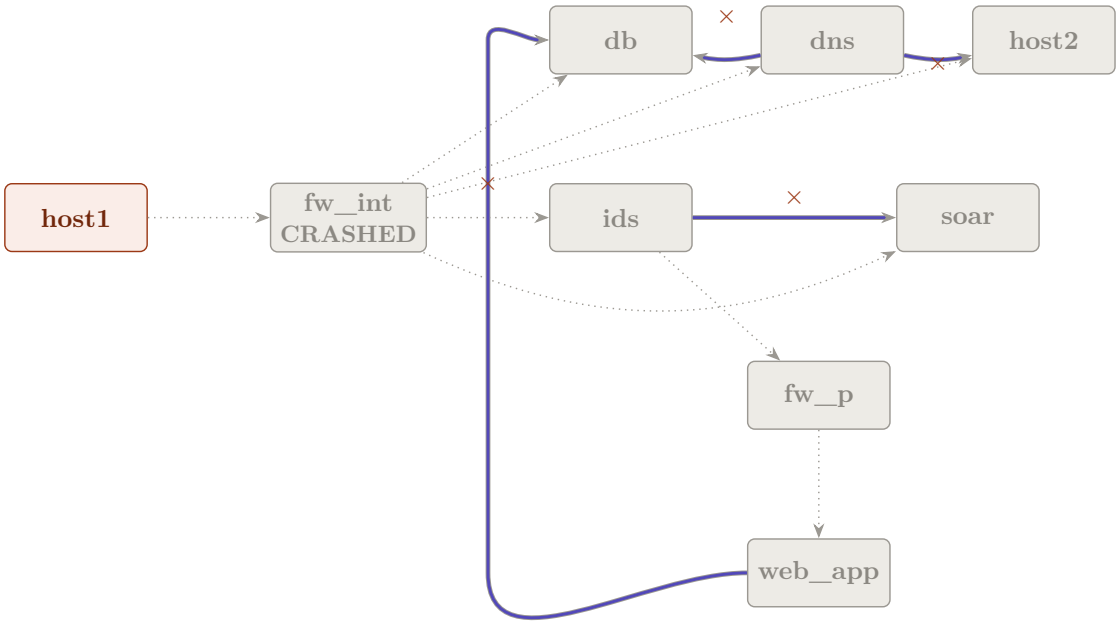


Figure 7.7: Example 2: the insider's perspective. The same crash seen from the case of `host1`. `fw_int` is `host1`'s only physical neighbor, so nothing remains reachable at all, not even `web_app`, whose only path from `host1` runs through `fw_int` and the sensor `ids`, nor `soar`, whose logical connection to `ids` is severed in the same way it was from `internet`. The lost dependencies are identical either way, since they depend on the topology of logical connections and not on who specifically caused the crash: the same single-chokepoint amplification of Section 5.3, here triggered by a real fault rather than a defense.

7.3.5 Playbook Risk

`playbook_a_rischio` reports which of the three case-study playbooks can be neutralized by an action that drives `soar` offline or leaves the analyst busy elsewhere, without the attacker's trace ever being touched. `pb_isolate_threat` lists `human_free` among its conditions, and so does `pb_incident_response_ransomware`; either action neutralizes both.

`alert_fatigue` is not a real playbook. It exists only to model one specific case, a sensor attack that drives `soar` offline directly (Section 6.5.3); that is its only purpose here.

7.3.6 Attacker's View Against Defender's View

Section 6.3 kept *Accepts* and *PolicyAccepts* as two separate predicates specifically to let this section ask a question a single, trust-parameterized predicate could not: does a defensive action still contain an attacker who already controls the very node the action targets, or does it only look like it does from the policy's own point of view?

`fw_p` answers this directly. It carries its own `Control`-impact vulnerability (Table 7.1), and the same pivoting rule of Section 6.5.2 that let `internet` reach `web_app` in Section 7.3.1 applies to it directly: `internet` has a physical link to `fw_p` and full control, so `fw_p` becomes compromised too. With `ids` blacklisted and `fw_p` compromised, `accepts(fw_p, ids, [block(ids)])` and `policy_accepts(fw_p, ids, [block(ids)])` were run against that identical state:

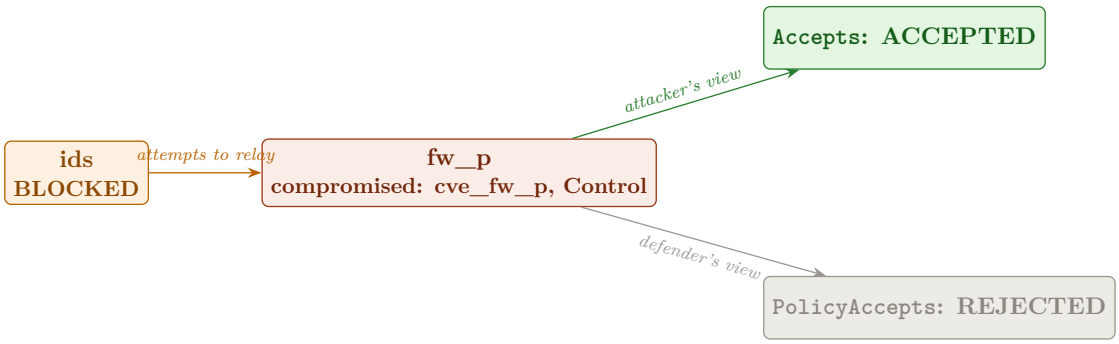


Figure 7.8: The same question, two different answers. Both predicates were run against the same hypothetical state: `ids` blacklisted, `fw_p` itself compromised with `Control` impact. The attacker's view accepts, because a compromised `fw_p` bypasses its own policy entirely; the defender's view rejects, because from policy alone the block looks like it worked.

The gap between the two answers is the entire point of keeping them separate.

`block(ids)` is a real, correctly recorded defensive action, and a policy-only audit of the network would report it as effective. It is not: `fw_p` stopped enforcing any of its own rules, whitelist or blacklist, the moment it was compromised, so the block it was supposed to carry out was never actually carried out. A model that only asked the policy question would have no way to notice this at all.

7.3.7 SOAR Manipulation

`asset_threat_surface(Target)` was introduced earlier for the routes that involve a compromise somewhere along the way. Its fourth and last section does not, but it is not just repeating the cascade rule already used a few times in this chapter: that rule never cared why a dependency ended up blocked, only that it did. This section shows a concrete way that block can happen without compromising anything at all, not even the node a fabricated alert blames (Section 6.6).

`asset_threat_surface(web_app)` finds three such routes, one for each node `web_app` depends on transitively, plus `web_app` itself:

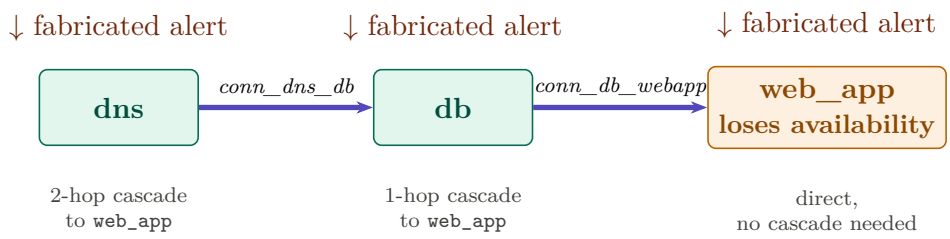


Figure 7.9: Three routes to the same loss, none of them a compromise. `asset_threat_surface(web_app)` reports all three: a fabricated alert can name `web_app` directly, or name a node it depends on at any depth in the chain, and the connection-cascade rule of Chapter 6 carries the resulting block the rest of the way to `web_app`. `dns`, `db`, and `web_app` itself are never compromised in any of the three; the SOAR's own trust in what a sensor reports is the entire attack surface.

A fabricated alert naming `dns` costs the attacker nothing beyond control of a single sensor, yet it reaches `web_app` two dependency hops away, exactly as far as compromising `dns` outright would. The engine draws no distinction in how far a false accusation travels compared to a true one: both are just a node entering `Malfunction`, and the connection-cascade rule of Section 6.5.1 does not ask how that happened.

Chapter 8

Conclusions and Future Work

The chapter 1 identified a gap that motivates this whole thesis: In other words, existing threat models treat the automation layer of the SOAR platform as an implicitly and always trustworthy observer of the network and not as a component whose own compromise, possible misconfiguration or simply overload is itself part of the attack surface. This closing chapter summarizes the contributions made, states the remaining limitations, and outlines potential directions for possible future work.

8.1 Summary of Contributions

The central contribution of this thesis is a formal threat model, presented in Chapter 6, that folds the SOAR itself into the same analysis applied to the network it protects. Two properties of the model make this possible, not just claimed. First, every predicate that governs routing exists in two versions, an attacker's view (*Accepts*) and a defender's (*PolicyAccepts*), kept structurally separate instead of collapsed into one predicate parameterized by a trust assumption; this is what later makes it possible to ask whether a defensive action still contains an attacker who already controls the node the action targets. Second, disruption has four independent, formally distinct causes rather than one: a native fault, a simulated crash, a dependency on a broken entity, and a dependency on a connection that policy alone no longer supports, this last one covering the case where a chokepoint the connection's physical route relies on has become unreachable regardless of whether the two endpoints it connects are themselves healthy.

That last rule is where the model's central empirical finding comes from. Chapter 5 argued, and Chapter 7 confirmed against the case study, that a chokepoint

shared by several independent logical connections turns a single failure into multiple, simultaneous losses of availability in services that share no relationship with one another and that were never themselves touched. The case study made this concrete, not just hypothetical: crashing `fw_int` breaks three logical connections at once, in services with no dependency on each other, and the same amplification pattern reappears as an explicit derivation rule rather than an emergent side effect, when the trigger is a fabricated alert instead of a fault, as Section 7.3.7 showed for `web_app`.

The model also formalizes failure modes specific to the automation layer that a network-only threat model has no vocabulary for, and a compromised sensor can fail the defense in three distinct ways. It can simply go silent on a trace it should have reported, since Section 6.5.3 excludes a sensor's own *Control*-impact compromise from counting as a witness on its own trace. It can flood the SOAR with fabricated, evidence-less alerts to induce saturation, the mechanism the case study calls *Alert Fatigue*. Or it can go further and name a specific, uninvolved entity, redirecting an automated containment action against a target the attack never touched, the mechanism this thesis calls *SOAR Manipulation* (Section 6.6). The case study confirmed that this last one can reach an asset several dependency hops away from the node actually named in the fabricated alert, at no extra cost to the attacker and with no entity in the chain ever compromised.

These rules were operationalized into a working Prolog engine (Chapter 7), validated against a representative case study through three complementary methods instead of trusting the engine's output at face value: manual derivation of a representative subset of results against the axioms directly, independent review by multiple readers, human and AI-based, each asked to propose edge cases the existing query battery might have missed, and differential testing across engine revisions to confirm that a fix changed only what it was meant to change. This process itself surfaced a real defect, the removal of a quarantine shortcut that a reviewer showed was unsound in the general case, and a methodological limitation in how the engine's own comparative queries conflate the effect of a defensive action with the effect of an entry point that was never authorized to begin with, a limitation the case study confirmed holds for both entry points no matter which node is chosen as the target of the defense.

Positioned against the closest methodological precedent, TAMELESS [12], this thesis does not propose a new formalism so much as extend an existing one, the assumed/derived convention TAMELESS established, to a domain that convention was not built to cover: a network where the response mechanism is itself an entity with its own vulnerabilities, its own dependencies, and its own capacity to fail in ways that matter to the analysis.

8.2 Limitations

The limitations of the model were stated as specific and deliberate simplifications when they were introduced for the first time in Chapter 6; they were a simple choice made in service of tractability and not for a generic oversight. Exploitation is evaluated as a binary outcome: if a vulnerability is present and a vector to it exists then the attack succeeds, with no representation of CVSS scores or success probabilities. In fact, the model reasons over the logical causality of events rather than their timing, verifying whether preconditions hold instead of estimating how long propagation takes. At most one physical link is assumed between any two nodes. This was done simply to avoid the algorithmic cost of computing paths over multigraphs.

The case study itself imposes a further boundary on what has actually been validated. Every result in Chapter 7 was checked against one topology, sized to be representative rather than exhaustive; the query battery covers every mechanism the model defines at least once, but a larger or differently shaped network could still surface an interaction the twenty-two queries used here did not happen to exercise.

8.3 Future Work

The most direct extensions follow from the limitations just stated. Replacing binary exploitation with a probabilistic model, closer to how CVSS scores are actually used in practice, would let the engine rank attack paths instead of merely enumerating them, at the cost of a search space that no longer collapses as cleanly as the current deterministic one does. Lifting the single-physical-link assumption to support real multigraphs would let the model represent redundant physical routes explicitly, which matters specifically for the chokepoint analysis of Chapter 5: a second physical path around a chokepoint is exactly the kind of mitigation the current model cannot represent as removing the amplification risk, because it cannot represent the redundant path to begin with.

A second direction extends the model's own conventions instead of lifting a restriction. *Offline* and *Busy* are checked and derived, in the current model, only for the single orchestrator and single analyst pool the case study declares; nothing in the derivation rules of Section 6.5.4 requires this, and a deployment with multiple SOAR instances or multiple analyst teams could drive the same two condition types against each independently, without changing the rules themselves. The quarantine shortcut, similarly, is written to be re-enabled, not rewritten, for any deployment where its precondition can be shown to hold.

A third direction is less about the model and more about where its results are

consumed. The engine currently answers queries posed by hand, one at a time, against a topology declared once at the start of a session; a natural extension is to close that loop, having the engine's own output, in particular a chokepoint's amplification radius or a detected SOAR manipulation risk, feed back into the same kind of automated configuration search that the network security automation literature of Chapter 2 already applies to firewall placement, so that a topology could be adjusted specifically to reduce a chokepoint's blast radius rather than merely reporting it.

Finally, every result in this thesis was obtained against a single, hand-built case study. Validating the model against a second, independently sourced topology, ideally one closer to a production deployment than the one used here, would test whether the mechanisms formalized in Chapter 6, in particular the chokepoint amplification pattern that motivated this thesis in the first place, generalize beyond the specific network built to demonstrate them.

Bibliography

[1] Daniele Bringhenti, Federica Pizzato, Riccardo Sisto, and Fulvio Valenza. «Autonomous attack mitigation through firewall reconfiguration». In: *International Journal of Network Management* 35.1, e2307 (2025) (cit. on pp. 3, 6).

[2] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, and Fulvio Valenza. «Automation for network security configuration: State of the art and research trends». In: *ACM Computing Surveys* 56.3, 57 (2024) (cit. on pp. 3, 5).

[3] Daniele Bringhenti, Guido Marchetto, Riccardo Sisto, Fulvio Valenza, and Jalolliddin Yusupov. «Automated firewall configuration in virtual networks». In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (2023), pp. 1559–1576 (cit. on p. 6).

[4] Daniele Bringhenti and Fulvio Valenza. «GreenShield: Optimizing firewall configuration for sustainable networks». In: *IEEE Transactions on Network and Service Management* 21.6 (2024), pp. 6909–6923 (cit. on p. 6).

[5] Daniele Bringhenti and Fulvio Valenza. «GreenShield-E2C: A sustainable energy-aware firewall configuration mechanism for edge-to-cloud continuum». In: *Computer Networks* 282, 112279 (2026) (cit. on p. 6).

[6] Daniele Bringhenti, Lucia Seno, and Fulvio Valenza. «An optimized approach for assisted firewall anomaly resolution». In: *IEEE Access* 11 (2023), pp. 119693–119710 (cit. on p. 6).

[7] Daniele Bringhenti, Simone Bussa, Riccardo Sisto, and Fulvio Valenza. «Atomizing firewall policies for anomaly analysis and resolution». In: *IEEE Transactions on Dependable and Secure Computing* 22.3 (2025), pp. 2308–2325 (cit. on p. 6).

[8] Daniele Bringhenti, Federica Pizzato, and Fulvio Valenza. «Improving web protection in virtual networks with automatic WAF configuration». In: *Proc. 2026 IEEE 12th International Conference on Network Softwarization (NetSoft)*. Berlin, Germany, 2026, pp. 556–562 (cit. on p. 7).

[9] Federica Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «An intent-based solution for network isolation in Kubernetes». In: *Proc. IEEE 10th International Conference on Network Softwarization (NetSoft)*. Saint Louis, MO, USA, 2024, pp. 381–386 (cit. on p. 7).

[10] Federica Pizzato, Daniele Bringhenti, Riccardo Sisto, and Fulvio Valenza. «Intent-driven network isolation for the cloud computing continuum». In: *Journal of Network and Systems Management* 34.1 (2026), pp. 1–39 (cit. on p. 7).

[11] Fulvio Valenza. *Course material for "Threat Intelligence" – 01NXEUV*. Politecnico di Torino (cit. on pp. 8, 9).

[12] Fulvio Valenza, Erisa Karafili, Rudolph Vieira Steiner, and Emil C. Lupu. «A hybrid threat model for smart systems». In: *IEEE Transactions on Dependable and Secure Computing* 20.5 (2023), pp. 4403–4417 (cit. on pp. 8, 10, 28, 57).

[13] Loren Kohnfelder and Praerit Garg. «The threats to our products». In: *Microsoft Interface* (Apr. 1999) (cit. on p. 9).

[14] MITRE Corporation. *MITRE ATT&CK*. URL: `https://attack.mitre.org` (visited on 2026) (cit. on p. 9).

[15] Tony UcedaVélez and Marco M. Morana. *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*. Wiley, 2015 (cit. on p. 9).

[16] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. «MulVAL: A logic-based network security analyzer». In: *Proc. 14th USENIX Security Symposium*. 2005, pp. 113–128 (cit. on p. 9).

[17] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. «A scalable approach to attack graph generation». In: *Proc. 13th ACM Conference on Computer and Communications Security*. 2006, pp. 336–345 (cit. on p. 9).

Acknowledgements

Desidero innanzitutto ringraziare i miei professori e i relatori che mi hanno seguito per la loro guida, la pazienza e il prezioso supporto accademico fornito durante la stesura di questa tesi.

Il ringraziamento più profondo va alla mia famiglia. A mio padre e mia madre, per aver creduto in me e avermi sostenuto incondizionatamente in ogni momento. Ai miei fratelli, per esserci sempre stati e per avermi dimostrato un affetto costante.

Un enorme grazie va agli amici del Camplus Bernini per aver reso questi anni indimenticabili. Un ringraziamento del tutto speciale è dedicato al mio compagno di stanza Adrian, che mi ha aiutato e tenuto compagnia, condividendo con me le sfide e le gioie di questa bellissima avventura.

Il mio pensiero vola poi agli amici di sempre, quelli che ho lasciato a Reggio Calabria: grazie di cuore per avermi riaccolto e fatto sentire a casa ogni singola volta che tornavo.

Infine, voglio ringraziare tutti i nuovi amici che ho avuto la fortuna di incontrare lungo questo cammino e che hanno reso questo viaggio ancora più speciale.