View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Electronic Thesis and Dissertation Archive - Università di Pisa

University of Pisa

and

Scuola Superiore Sant'Anna

Master Degree in
Computer Science and Networking

A Simulation Based SIEM Framework to Attribute and Predict Attacks

Candidate

Jacopo Lipilini

Supervisor

Fabrizio Baiardi

Abstract

We present a Security Information and Event Management (SIEM) framework to correlate, attribute and predict attacks against an ICT system.

The output of the assessment of ICT risk, that exploits multiple simulations of attacks against the system, drives the building of a SIEM database. This database enables the SIEM to correlate sequences of detected attacks, to probabilistically attribute and predict attacks, and to discover 0-day vulnerability.

After describing the framework and its prototype implementation, we discuss the experimental results on the main SIEM capabilities.

Contents

1 Introduction **1**
1.1 Intrusion Prevention and Detection 1
1.2 Thesis goals 2
1.3 Thesis structure 3

2 Related works **5**
2.1 Intrusion Detection Systems 5
2.1.1 Types of Intrusion Detection Systems 7
2.1.2 Detection methods 10
2.2 Security Information and Event Management 11
2.2.1 Alerts correlation 14
2.3 The Haruspex suite 19
2.3.1 Kernel modules 20
2.3.2 Attacks 21
2.3.3 Agents 22
2.3.4 Simulations 24

3 Framework **25**
3.1 Architecture overview 25
3.2 Sensors 26
3.3 Failure detection case 28
3.3.1 Building the SIEM database 28
3.3.2 Receiving and filtering alerts 29
3.3.3 Agent attack patterns recognition 31
3.3.4 Correlating attacks 33
3.3.5 Attributing and predicting attacks 36

3.3.6 Discovery of 0-day 40
3.4 Attack detection 42

4 Implementation 46
4.1 Data structures 46
4.1.1 Pattern Pool 46
4.1.2 Pure Sequence Trie 47
4.2 Algorithms 50
4.2.1 Pure sequence database construction 50
4.2.2 Agent pattern set construction 51
4.2.3 Agent pattern matching 53
4.2.4 Correlation 53
4.2.5 Attribution and Prediction 57
4.2.6 Investigation 59

5 Experimental results - Failure Detection 61
5.1 Testing environment 61
5.1.1 Synthetic database 62
5.2 General information 63
5.3 Pattern matching capability 63
5.4 Attribution capability 67
5.5 Prediction capability 69

6 Experimental results - Attack Detection 71
6.1 Testing environment 71
6.2 General information 71
6.3 Pattern matching capability 72
6.4 Attribution capability 74
6.5 Prediction capability 75

7 Conclusions 77
7.1 Final remarks 77
7.2 Future works 79
7.2.1 Real environment test 79

7.2.2 Sensors deployment and ruleset generation 79
7.2.3 False positive and false negative handling 80

Bibliography **81**

Chapter 1

Introduction

1.1 Intrusion Prevention and Detection

Modern ICT systems rely on a complex infrastructure that strongly increases the complexity of the security monitoring.

While current *Intrusion Detection and Prevention Systems* consider the attacks in isolation, the attack scenarios they have to face are rather complex. As an example, several tools that are freely available can be used to develop malware to automatically attack system nodes from another node they have already attacked. These tools can produce code that can result in very large impact in a short time.

One of the most complex case intrusion prevention and detection system have to face is the one of intelligent threat agents. These agents attack an ICT system to achieve a predefined goal. As an example, to steal some information stored in a given node of the infrastructure. To reach any of its goals, the agent has to acquire a set of access rights on the system components. When achieving a goal, the agent may violate any of three main security properties: integrity, confidentiality and availability. The first and the second one refer to, respectively, the unauthorized updates and access to information, while the third to a denial of service. The main problem an agent has to solve to achieve a goal is that a single attack may not grant all the rights in a goal. As a consequence, the agent has to implement a sequence of attacks.

In the described scenario, current *Intrusion Detection and Prevention Systems* may fail to protect the target system because they do not return information to correlate attacks and discover those that belong to the same sequence.

This justifies the actual trend in implementing *Security Information and Event Management* (SIEM) tools. This new security tool should rebuild a complete and reliable security status of the system by collecting and correlating the output of a large network of sensors, where each sensor detects single attacks against system components. A SIEM system should not only enable a security administrator to discover and react to attacks in real-time but also to discover network misconfiguration, weaknesses, and system updates.

Despite the large number of available tools and their capabilities, the automation of the integration and of the correlation of data from a sensor network is still an open problem.

These challenges and their impact on security are the main motivations of this thesis that defines and evaluates a new SIEM framework to correlate, attribute and predict attacks against an ICT system.

1.2 Thesis goals

This thesis evaluates an overall framework to solve the problems previously outlined. These problems are the correlation, attribution and prediction of attacks against the target system.

The *correlation* is the problem of how to interpret a set of alerts to pair it with a proper meaning. An alert is the alarm raised by a sensor when it detects an attack. A proper meaning is the discovery of the attack sequence that an agent implements to escalate its privileges.

The SIEM needs a proper knowledge base to correctly correlate alerts. This knowledge can be acquired from several sources. One of them is the output of a tool to automatically assess the risk of an ICT system. This tool returns, among others, a database with the attack sequences each intelligent agent implements against the target system to reach its goal.

By exploiting the risk and vulnerability assessment to drive the building

of the SIEM database, we can increase the correlation capability of the SIEM, because it is aware of the attacks the agents may implement. Furthermore, a mismatch in the correlation may signal a possible 0-day vulnerability, i.e. a vulnerability that is not public yet, or an unknown update to the target system.

The correlation of alerts supports the *attribution* and *prediction*. The former consists in the identification of the agent that is currently implementing a detected sequence, while the latter consists in the forecast of the next attacks. The solutions to these problems simplify the discovery of the goal the agent aims to achieve and of its next attacks. The attribution of a sequence of attacks and the prediction of the next one constitute an important mechanisms to minimize or completely avoid the impact of attacks.

Another approach we evaluate to identify attacking agents relies on the generation of unique attack combinations, the *patterns*. Patterns are generated by analyzing the attack sequences each agent implements. The set of patterns is then matched against the alert stream to identify the corresponding agent.

After implementing a SIEM framework with these features, we evaluate its main capabilities. In particular, we focus on the evaluation of the reliability and accuracy of the pattern matching, attribution and prediction.

1.3 Thesis structure

In addition to this chapter, the thesis is structured as follows.

Chapter 2 This chapter presents a survey of the current state-of-the-art on Intrusion Detection System and Security Information and Event Management. It also introduces the Haruspex suite that supports the automated assessment of ICT system.

Chapter 3 This chapter outlines the proposed framework and the main problems to correlate, predict and attribute attacks. It also discusses the solutions we propose to these problems. This section also introduces the definitions to formalize the framework.

Chapter 4 This chapter discusses the prototype implementation of the framework. We detail the main algorithms we have developed and the data structures they exploit.

Chapters 5 and 6 These chapters report the experimental results to evaluate the main SIEM capabilities. In particular, we evaluate the agent identification and the prediction capabilities.

Chapter 7 This last chapter resumes the main results of the thesis and outlines future works.

Chapter 2

Related works

This chapter reviews some concepts and works related to intrusion detection, security information, event management, and risk assessment.

2.1 Intrusion Detection Systems

An *Intrusion Detection System* (IDS) is a device or software module that monitors a system to detect attacks or policy violations. The monitoring phase is implemented by means of *sensors*, entities that check events to discover violations according to the approaches described in the following. When these sensors detect a violation, they raise an *alert* with information relative to the event. These concepts were firstly introduced in [5] and then again formalized in [18]. A working group created by DARPA in 1998 defined a common framework for the IDS field [41] and a common intrusion specification language about attacks and events to enable cooperation among the components of an intrusion detection and response (ID&R) system.

A very simple IDS architecture is discussed in [17], depicting it as Figure 2.1. Basically, an IDS is a detector that processes information coming from the system to be protected. Three kinds of information are used. The first one is long-term information related to the technique to detect intrusions (a knowledge base of attacks, for example). The second kind of information is configuration information about the current state of the system. The third

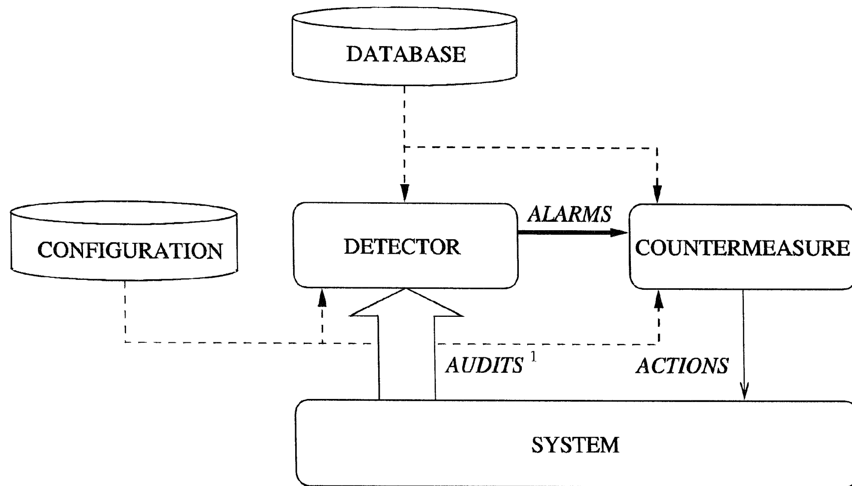

Figure 2.1: A simple IDS architecture

and last kind of information is audit information that describes the events that may occur in the system. The role of the IDS is to eliminate useless information from the audit trail and present a synthetic view of security-related actions by the users. A decision is then made to evaluate the probability that these actions can be considered symptoms of an intrusion.

According to [6], several reasons support the adoption of an IDS such as preventing attacks, increasing the perceived risk of discovery and punishment of attackers, documenting the existing threat, and detecting the preambles to attacks.

The IDS has to perform its task in real-time to be beneficial from a security perspective. In principle, an IDS has to deal with any kind of intrusion, like network attacks against vulnerable services, data driven attacks on applications, host based attacks such as privilege escalation, unauthorized logins and access to sensitive files, and malware (denial of service attack, viruses, trojan horses, and worms). Alternative kinds of sensors analyze different data and apply distinct detection methods to deal with the distinct features of an intrusion [27, 37, 46].

2.1.1 Types of Intrusion Detection Systems

We can distinguish between *network IDS* (NIDS) and *host IDS* (HIDS).

NIDS can be further partitioned into two subtypes, the wireless IDS, focusing on wireless network, and the network behavior analysis (NBA) IDS, examining traffic flow on a network in an attempt to recognize abnormal patterns like Distributed Denial of Service (DDoS), malware, and policy violations.

Another kind of IDS of interest is a *distributed IDS* (DIDS) that combines HIDS and NIDS to build an efficient and cooperative security environment to acquire a broader view of the whole security status of the system. One of the first examples is [40].

Network IDS

Since a NIDS monitors the network traffic, it is important that it can inspect most of inbound and outbound network traffic. Ideally, all the traffic should be analyzed, but this could impair the overall network performance. Each packet is captured by a sniffer (both hardware and software solutions are available) and then it is analyzed to detect possible attacks. This exploits a special implementation of the TCP/IP stack that reassembles the packets and applies protocol stack verification, application protocol verification, or other verification techniques.

In the protocol stack verification, the NIDS looks for malformed data packets that violate the TCP/IP protocol. This process is useful mainly to detect DoS or DDoS attacks, because they rely on the creation of improperly formed packets to exploit any weaknesses in the protocol stack.

The application stack verification considers rules of higher-order protocols, like HTTP, to discover unexpected packet behavior or improper use. One example of this kind of attack it can discover is DNS cache poisoning. This verification process requires more computation time than the previous one, so it could affect the NIDS performance.

The main advantages of NIDS include:

- if the network is well designed, few NIDSs can monitor a very large

network,

- NIDS, as passive devices, can be deployed with little or no disruption to normal network operations,
- again as passive devices, they are usually not susceptible to direct attacks and not detectable by attackers.

The disadvantages are:

- because of increasing network bandwidth, a NIDS can be overwhelmed by network traffic, compromising also its detection capabilities,
- the NIDS effectiveness is limited by encrypted communications and fragmented packets,
- NIDS cannot reliably discovery whether an attack was successful.

One of the most adopted NIDS tool is Snort [31], a free and open source network intrusion detection (and prevention) system, running on most modern operating systems. Furthermore, it is supported by a large community. Snort performs real-time traffic analysis and packet logging on Internet Protocol (IP) networks, doing protocol analysis, content searching, and content matching. Snort integrates distinct components. These components cooperate to detect particular attacks and to generate output in a required format. There are five main components. The *Packet Decoder* receives packet from different types of network interfaces. Different *Preprocessors* are used to normalize protocol headers, detect anomalies, packet reassembly and TCP stream reassembly for the next detection phase. The *Detection Engine* detects intrusion activity in packets. The *Logging and Alerting System* generates alerts and logs. Finally, the *Output Modules* generate the final output from the previous results.

Host IDS

The HIDS monitors a specific host or device on the network. Usually, in the monitored host, the HIDS is installed as a dedicated software or hardware that analyzes local events. The HIDS is also known as *system integrity*

verifier because it monitors the status of key system files and detects when an intruder creates, modifies, or deletes a monitored file. The HIDS examines these files and system logs to determine if an attack is underway or has occurred and if the attack was failed or successful. Since distinct priority levels are associated with distinct resources, the most common method to categorize folders and files is by *color coding*. Red coded resources, like OS kernel, are the most critical one. The yellow coded, like device driver, are less critical, while the green coded resources, like user data, are less urgent, because they are frequently modified.

The advantages of HIDS include:

- it can detect local events on host systems and attacks that may elude a NIDS,
- it can process encrypted traffic, because it is decrypted by the host,
- it can detect inconsistencies in the use of applications and of system programs. This enables the detection of Trojan horse,
- it can detect success or failure of attacks with respect to NIDS.

As a counterpart, the main disadvantages are:

- more management is needed, because HIDS are configured and managed on each monitored host,
- it is vulnerable both to direct attacks and to those against the host operating system,
- it needs large amounts of disk space to store the host OS audit logs,
- it can noticeably affect the performance of its host systems.

A widely used HIDS is OSSEC [12]. It is free and open source, and provides intrusion detection for most operating systems. It performs log analysis, integrity checking, Windows registry monitoring, rootkit detection, time-based alerting, and active response. OSSEC has a centralized, cross-platform architecture allowing multiple systems to be easily monitored and

managed, such as databases (like MySQL), web servers (like Apache HTTP Server), firewall (like Iptables), NIDS (also Snort), and many others. It is composed by three components. The *Main Application* is required for distributed network or stand-alone installations. The *Windows Agent* monitors Windows environments. The *Web Interface* provides a graphical user interface.

2.1.2 Detection methods

When an IDS analyzes network traffic or local events to detect malicious activities, it can apply three main strategies: *signature based*, *anomaly based* and *stateful packet inspection* detection.

Signature based detection

Signature based approaches detect attacks by matching their input against a database of signatures of known intrusions. As a consequence, the attacks are signaled in a fairly accurate way.

The signature based detection method is widely used because many attacks have unambiguous and distinct signatures. Consider, for example, fingerprinting activities and worms. They implement one of a small set of attack sequences designed to exploit some vulnerabilities and control a system.

The advantages of this approach are the ease of writing a new signature and of understanding signature others have developed. Obviously, this assumes that enough information on possible attacks is available. Further advantages include a very precise notification of the events that caused the alerts and the ability of simplifying the signature database by disabling rules not needed. As an example, rules for SMTP traffic are not enabled if an administrator knows that this traffic is not utilized.

Drawbacks include the need to gather the most comprehensive set of information on an attack to extract a accurate signature and the frequency of updates to the rule database. This method cannot detect a completely new vulnerability and so its exploit, a so called *0-day attack*.

Anomaly based detection

Anomaly based detection is an approach based upon statistics that assume that we know the normal behavior for a node or network. Under this assumption, detection is implemented by comparing statistical indicators, like the volume of network traffic or CPU usage, against the normal behavior.

The main advantage is the detection of *0-day attack* if they result in a behavior that fall out of the interval of normal statistics. Another advantage is the ease of adaption that only requires the update of the thresholds rather than the definition of new signatures.

The drawbacks are the high number of false positives, events signaled as malicious while they are not, and the complexity to collect data to statistically define the *normal* behavior.

Stateful packet inspection approach

Assuming that the IDS knows how a protocol, such as FTP, is supposed to work, it can detect anomalous behavior.

Relevant data per session are stored and then used to identify intrusions that involve multiple requests and responses. This approach can also detect multisession attacks. The Stateful Protocol Analysis (SPA) examines packets at the application layer for information extracting. This is also referred as *deep packet inspection*.

The main drawback is the complexity of session based detection, as it introduces both heavy processing and memory overhead to track multiple simultaneous connections.

2.2 Security Information and Event Management

A *Security Information and Event Management* (SIEM) is a component that implements real-time analysis of the alerts generated by sensors. In some sense, it is the evolution of an IDS.

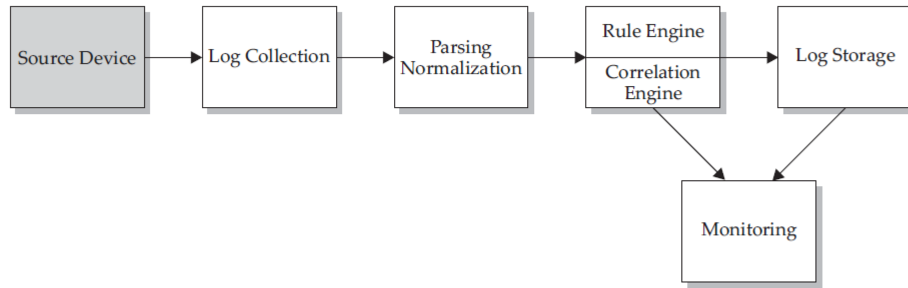

Figure 2.2: The general SIEM architecture

As the name suggests, a SIEM combines the aspects of *Security Event Management* (SEM) and the *Security Information Management* (SIM). The former deals with real-time monitoring, correlation of events, notifications and console views. Instead, the latter provides long-term storage, analysis and reporting of log data.

The general architecture of a SIEM is discussed in [28] in terms of the Figure 2.2. The first part of a SIEM is the source device that feeds information into the SIEM. A source device is the device, or the application, that supplies logs to store and process in the SIEM. A modern SIEM can cooperate with different source devices, even from different vendors. Actually, a log collection process retrieves different logs in two ways. Either the source device sends its logs to the SIEM (*push method*), or the SIEM reaches out and retrieves the logs from the source device (*pull method*). Because logs are still in their native format, they have to be parsed and normalized in a single format. Now the rule engine analyzes the normalized logs events to trigger alerts due to specific conditions. The correlation engine is a subset of the rule engine that matches multiple standard events from different sources to produce a single correlated event. To take into account the volume of logs that the SIEM receives, usually the logs are stored in a database, a flat text file or a binary file. Finally, the SIEM implements a method, web-based or application-based, to enable the user to interact with the logs.

[25] describes a framework for attack modeling and security evaluation in SIEM systems. This framework includes several steps:

1. the modeling of malicious behavior,

2. the generation an attack graph,
3. the evaluation of distinct security metrics,
4. the definition of risk analysis procedures.

The key elements are the use of a comprehensive security repository, and of an effective attack graph (tree) generation techniques (also in near-real time). Furthermore, it takes into account known and new attacks based on 0-day vulnerabilities, and it supports stochastic analytical modeling, and interactive decision support to select the most effective security solutions. The defined Attack Modeling and Security Evaluation Component (AMSEC) can behave in two modes. In non real-time mode, AMSEC produces the list of weak network places, possible 0-day vulnerabilities, and the set of attack trees. This output is computed through the model of the computer network. In turn, this model is defined according to the design specifications as well as to the network configuration and security policy. In the real-time or near real-time one, the AMSEC adjusts existing attack trees and malefactor model to predicts actions of an attacker and generate countermeasures.

One of the most popular SIEM tool is OSSIM [23]. It is an open source SIEM, that integrates a selection of tools that support network administrators in computer security, intrusion detection and prevention. OSSIM provides integration, management, and visualization through a browser-based user interface of events of open source security tools, like Snort and OSSEC. More important, OSSIM simplifies the integration of new security devices and applications. After the normalization of the collected alerts, OSSIM applies event filtering and prioritization through configurable policies. OSSIM applies three types of correlations. *Inventory Correlation* filters attacks to specific kind of asset, e.g. Windows threat to Linux box. *Cross Correlation* compares event and vulnerability analysis results. *Logical Correlation* correlates using user defined condition trees.

A core task of a SIEM is the correlation among alerts to understand what is happening in the system to inform a security expert, deploy some countermeasures, and so on. A general characterization of the correlation

process is explained in [43], and refined in [36]. We can distinguish four phases. At first, the alert enters the *preprocessing phase* for the normalization and enrichment with other useful information for the next phases. Then, the *reduction phase* filters and validates the alert. The third step is the *correlation phase* that tries to find out how an actual alert is related to the previous ones. The fourth step, the *prioritization phase*, ranks the previous results according to their severity.

During the correlation phase, the SIEM tries to discover the meaning of the alerts stream received so far. A possible way to explain alerts correlation, as in [30], is the generation of *hyperalerts*. These abstractions are basically concrete attacks to the system and compose a *correlation graph*, representing the different attack scenarios. Different techniques can be used to find a causal relation between hyperalerts, such as temporal constraints, but more important, the *cause-effect* one, expressed as prerequisites and consequences of each hyperalert.

In the following we describe different approaches for alerts correlation, because it is the kernel of the intelligence of every SIEM tool.

2.2.1 Alerts correlation

In last years, a large amount of research efforts focused on alerts correlation techniques and their taxonomy [11, 35, 36, 49, 51].

Several definitions of alert correlation can be found in the literature [19, 21, 33], but all these definitions basically describe alert correlation as the *interpretation of multiple alarms to pair them with a proper meaning.*

Alerts, also referred as alarms, are generally short textual messages in a specific format defined by vendors or by a standard, like the IDMEF [16]. They are generated on the basis of a matching between some predefined rules and network or host events. Typically, alerts contain general information regarding the device issuing them, e.g. its IP, and the event itself, e.g. the creation time, a description of the event, references to vulnerabilities database, impact, and so on.

Several reasons favor the adoption of alert correlation. The most impor-

tant one is the need of discovering the root causes of a problem. An example of a root cause is an initiating cause of an attack chain [22].

Alert correlation techniques are used in three different application domains: network management, industrial process control (SCADA system) and network and system security. Obviously, here we focus on this last domain.

In the field of network and system security, alerts are generated by security elements such as NIDS and HIDS. Since the sensitivity of the detection process is highly variable, they could generate a huge amount of alarms, where some of them only signal normal activity rather than attacks (*false positives*). Alerts correlation simplifies the evaluation of the validity of those alerts, and, more important, the detection of complex and multistep attack scenarios. Usually, this results in a comprehensive view on the security state of a system.

Alert correlation can use several sources of information, besides the alerts itself. For instance, we recall topology information [14] and vulnerabilities databases [47].

Focusing on distinct strategies to correlate alerts, in the literature we find three main category, *similarity-based*, *sequential-based* and *case-based* methods.

Similarity-based methods

Similarity-based methods try to cluster and aggregate alerts using their similarities in attributes such as the IP addresses, port, protocol and timestamp information. We can distinguish between *attribute* and *temporal* based categories, depending on how similarity is computed. Attribute based techniques correlate alerts by computing predefined metrics, such as Euclidean, Mahalanobis, Minkowski, and/or Manhattan distance functions on some attributes. The resulting scores are compared to a threshold to determine if alerts have to be correlated. Instead, temporal based techniques rely on timing constraints.

In [42] a probabilistic method is proposed. An appropriate similarity

function is defined for each attribute in a range from zero (mismatch) to one (perfect match). The overall similarity is computed through an equation that combines the results of the previous functions. For each new alert, the similarity is computed for existing meta-alerts, and the newly created alert is merged with the best matching meta-alert, as long as the similarity is larger than a threshold value. Otherwise, a new meta-alert is created.

[15] uses a distinct approach, an expert system one, that defines the similarity relationship in terms of requirements, each specified through expert rules. These rules are domain specific and they are defined through an analysis of the alerts generated by distinct IDSs. The rules are partitioned into four categories based on alerts attributes. These are classification, time, source and target of the alert.

[1] uses several time windows, along with a trained classification method, to avoid comparing new alerts against the whole set of received alerts. Then, alerts are correlated though a probability estimation function. Two alerts are correlated if their temporal similarity is higher than a threshold.

The well-know concept of entropy is the basis of [20]. For each alert, the partial entropy is calculated to find the alert clusters with the same information. Alert clusters are represented by hyper-alert. Finally a subset of hyper-alerts is selected according to the entropy maximization.

The similarity-based method is the simplest one and it can be implemented through simple and lightweight algorithms. However, the important drawback is that it cannot detect root causes. The identification of a root cause is very useful, because it is an initiating cause of an attack chain. Hence, we can prevent further occurrences of the attack chain by removing its root cause [22].

Sequential-based methods

Sequential-based methods group alerts according to causality relationships, represented as a logical formula using combinations of logical operators, on *pre-conditions* and *consequences* of attacks.

A large number of solutions has been proposed, using several approaches

to model a scenario. We can mention, as an example, *pre/post conditions*, *(attack) graphs* and *(Hidden) Markov models*.

In the pre/post conditions category we recall MARS [3, 4]. Attack consequences are modeled through vulnerability and extensional consequences. The latter is an extended description of possible consequences in a form of predicates with free variables of facts, such as IP addresses. At first, raw alerts are normalized and then aggregated. Instances of multi-stage attack instances are generated by correlating aggregated alerts. The proposed approach is a variation of the requires/provides model and considers five factors to determine the link between stages of attack sequences. These factors are temporal (alert timestamps) and spatial (IP addresses and port) relationships, pre and post conditions of attacks, vulnerability assessment of the target system and its configuration.

[44, 45] use an attack graph approach. The correlation is based on a Queue Graph, which has the ability to hypothesize missing alerts and to predict future alerts. It only keeps in memory the latest alert matching for well-known exploits. The correlation between a new alert and the in-memory ones is explicitly recorded, whereas the correlation with other alerts is implicitly represented in terms of the temporal order between alerts. This improves the overall efficiency as the correlation process does not need to scan all the previously received alerts.

An interesting hybrid model is proposed in [2]. It consists of two parts. The main one applies an attack graph-based method, extending [45], to correlate alerts raised for known attacks and hypothesize missed alerts. Instead, the second one uses a similarity-based method, based on [1], to correlate alerts raised for unknown attacks which cannot be correlated by the first part. The novelty of the approach is the capability of hypothesizing missed exploits and of discovering defects in pre and post conditions of known exploits in attack graphs. It can also update the attack graph by applying the similarity-based method in the second part of the model.

An interesting combination of attack graph and Hidden Markov Model (HMM) is used in [34], that presents a formal model of the correlation algorithm. The algorithm can be parameterized to tune its robustness and

accuracy. Two approaches improve the speed and quality of the algorithm. Firstly, a parallel multi-core version, using both CPUs and GPUs, is proposed. Then, on a HMM-supported version, the Viterbi Path algorithm is computed to identify the most probable path in the corresponding attack graph. The correlation platform can work in real-time.

[48] uses the Hidden Markov Models to represent typical attack scenarios. It includes an online tracking and prediction module and an offline model-training module. The former searches the best attack scenario to describe the alert sequence and finds the most likelihood state transition (attack intention), while the latter uses the historical alert data to build the Hidden Markov Models for the typical attack scenario.

[39] uses a Markov chain to build a probabilistic model of abnormal events in network systems to forecast and detect network intrusions. It consists of three phases. In the first phase, the network states, including the outlying ones, are newly defined by applying a K-means clustering over a training data set. Based on these states, the second phase computes the state transition probability matrix and the initial probability distribution of the Markov model. The third phase computes in real-time the chance of abnormal activity for online data.

The main advantage of the sequential-based method is the high accuracy in recognizing attack scenarios, potentially discovering the causal relationship between alerts. However, the correlation results depends upon the logical predicates that are defined as well as upon the quality of the sensors.

Case-based methods

Case-based methods rely on a knowledge-base system that represents well-defined scenarios. The underlying knowledge is built by human or inferred by adopting machine learning or data mining techniques, and is continuously updated with the new observed scenarios. When a new case is raised, the system searches the database for the most similar cases. If a matching case is found, its associated information is retrieved and used to solve the problem. If this attempt is successful, proper information on the solution is stored

for future reuse. Otherwise, the reasons for the failure are identified and recorded for the next decisions.

A methodology and language for modeling multistep cyber attack scenarios is proposed in [13], that models scenarios as trees. Each scenario is represented through a set of modules. A module represents an inference step and consists of three sections: activity, pre and post condition. To support event-driven inferences, the activity section specifies a list of events to trigger the module. A library of predicates is defined and used as a vocabulary to describe the properties of system states and events. Each module is linked to others through pre/post conditions to recognize attack scenarios.

Some proposals adopt an approach based upon data mining, a set of techniques and tools to extract and present implicit knowledge. [24] presents a method to discover, visualize, and predict behavior pattern of attackers in a network based system. Data mining techniques are applied to generate association rules, starting from alerts produced by an IDS, and to build predefined attack scenarios. These scenarios are used to predict multistage attacks.

Case-based correlation techniques are very effective to solve well-known problems by specifying appropriate solutions and by discovering new potential ones. The drawbacks are the complexity of building an exhaustive list with all the scenarios to build a comprehensive knowledge database and the low performance that prevents their adoption for real-time correlation.

2.3 The Haruspex suite

The Haruspex suite [7, 8, 9, 10] is a collection of tools that enable security experts to model threat agents, simulate attacks and choose countermeasures for a system. Starting from the list of all the vulnerabilities affecting each node and from the system topology, a module of the suite builds an internal model of the system. A number of experiments on agents attacks are implemented through this model by applying a Monte Carlo method. By collecting proper information in each simulation, a statistical sample is built and stored in a database. This database stores information on the attacks

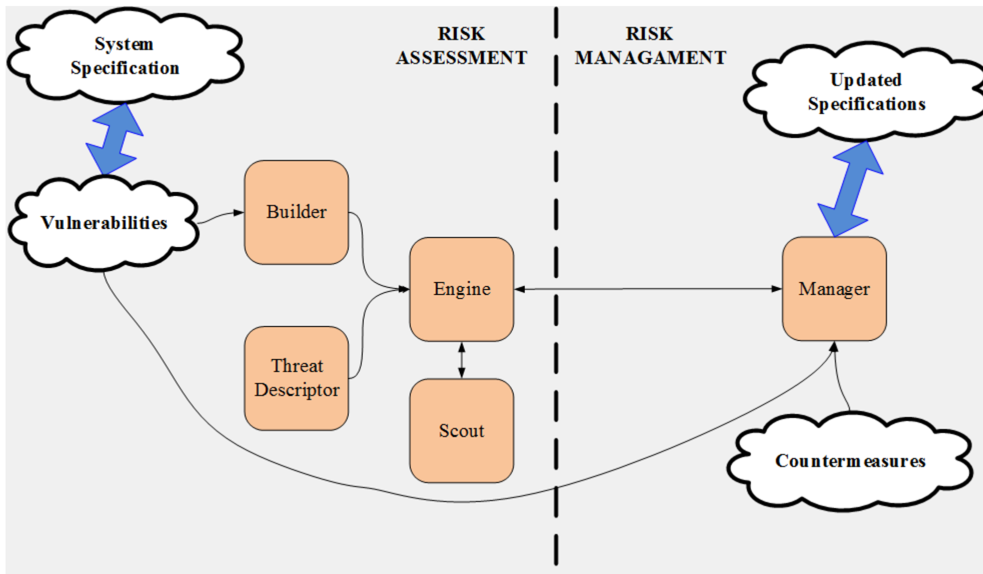

Figure 2.3: The architecture of the Haruspex suite

these agents have implemented, the goals they have reached and the time this has taken. Other tools use this database to produce statistics to assess the risk and select countermeasures to be deployed.

The suite introduces several definitions and features. Here we detail just some of them.

2.3.1 Kernel modules

The Haruspex suite is composed by several modules, as depicted in Figure 2.3, but the kernel ones are the *builder* and the *engine* module.

The *builder* module

The *builder* module creates the model of the system, starting from the output of one or more vulnerabilities scanners, the vulnerabilities the user suspects will be discovered in the future and the logical system topology. Using this information, an enumeration of all the elementary attacks to the system is made, enriched with other information described in the following. Each vulnerability is classified using the Common Vulnerabilities and Exposure

(CVE) database [29, 32], a de facto standard on vulnerability description, and the Common Vulnerability Scoring System (CVSS) [38]. The information extracted by the *builder* are used by other modules of the suite.

The *engine* module

The *engine* module implements a Monte Carlo method to produce an assessment database with the statistical sample to support the assessment. Starting from a description of the attacks to the system previously discovered and one of the agents, a number of experiments are made to simulate the possible attack chains that the agents can implement against the system. This number is specified by the user or is determined on the basis of a user defined confidence interval on the statistics. Further details on this phase are explained in the following.

2.3.2 Attacks

An *attack* is the exploitation of a *vulnerability* of a component by an agent to gain new rights or shutdown a service. A component represents a hardware or software module of the system. A vulnerability is a defect in a component or an erroneous or malicious behavior of some users. The considered vulnerabilities are *effective*, if already discovered, and *potential*, if they are only suspected, so they are paired with the probability distribution of being discovered at a given time.

An attack is characterized by the target IP, port and the considered protocol. A further attack attribute is the vulnerability identifier, in the form of a CVE id. From the vulnerability description in the CVE, the *builder* extracts further attributes of the attack, such as the success probability, the prerequisites and the consequences. These last two attributes represent respectively the rights needed to implement the attack and the rights acquired if the attack is successful.

The attack previously considered is an *elementary* one. Instead, a *complex* attack is the composition of elementary attacks, an attack chain. The composition has to respect some constraint. First of all, an elementary at-

tack is never repeated after its success. Then, the first attack preconditions must be included in the initial rights set of an agent. The last constraint is that a chain should respect attack pre and post condition. This means that the rights returned by an initial subsequence of the elementary attacks in the chain define a set of rights that includes the preconditions of a successive attack.

2.3.3 Agents

The attackers that try to violate the system are modeled as *threat agent*, or simply *agent*.

The suite considers an intelligent agent that compose elementary attacks into a complex one to reach some *goals*. Each goal is a distinct set of *rights* that the agent reaches after acquiring all the corresponding rights. Each right enables the agent to invoke the corresponding operation of a resource, like turning off a service, read or write record of a database, being user of a node, and so on. Obviously, when an agent reaches a goal there is an *impact*, that is a loss for the owner of the system.

The user specifies the model of the agent through *attributes* like the initial set of rights and resources available, and the set of rights that the attacker wants to acquire.

Obviously, to effectively model an intelligent attacker, the *engine* module has to simulate the possible attack choices that the agent could implement. To do so, the *engine* uses an *attack graph*. Every node of this graph represents the rights acquired by the attacker when an attack is successful. We can also distinguish the *initial node*, the starting point for a chain, and the *final node*, containing the goals of the agent. The path in the graph from an initial node to a final one basically represents a privilege escalation to reach a goal. This path also represents the complex attack that the agent has to successfully implement in order to achieve some goals.

There are also other attributes to better refine the simulation of the most real behavior of the attacker, by changing the selection of the next attack of an agent.

One of the most important ones is the *ranking strategy*. The agent applies this strategy to select the sequence of attacks to implement. The strategy considers attributes such as the success probability, the time to implement the attacks or the number of rights it grants to the agent. Among the various strategies an agent can apply, four are the main ones. The *Max Probability* one selects the complex attack with the higher probability of success. The *Max Increment* one selects the complex attack that returns, if successful, the largest set of new rights. The *Max Efficiency* one selects the complex attack with the best ratio between success probability and execution time. The *Smart Subnet First* one selects with same probability each elementary attack, but it assigns a larger priority to attacks that returns rights on another subnet.

The interesting notion of *look-ahead*, a positive integer expressing the quantity of information that an agent uses when has to choose the next attack to implement, is defined. Typical values are 0, 1 and 2. The next attack is chosen by considering paths of look-ahead length starting from the node an agent has reached in the attack graph. Obviously, the look-ahead value influences the previously defined ranking strategies. In particular, two cases may arise. If any ranked attack grants the rights in a goal, then it is returned. Instead, if, due to a low look-ahead value, the strategy cannot determine a complex attacks that leads to a goal, then the selection is done according to the ranking strategies. In this last case, the agent may implement useless attacks.

The *persistence* of the attacker is the number of times a failed attack is repeated before selecting another one.

The *continuity* is the number of attacks of a chain the agent executes before invoking again the strategy. This defines the compromise between the selection overhead and the ability of executing chains enabled by newly discovered vulnerabilities.

2.3.4 Simulations

Coherently with the adoption of the Monte Carlo method, an *experiment* includes several, different and independent *runs*. Each run simulates the behavior of some agents for the same time interval, together with the discovery of potential vulnerabilities. A run is subdivided into time steps to analyze these aspects in the simulated elapsing time. When a run is over, the *engine* module re-initializes the status of the system and agents to start a new, independent, run.

At each time step of a run, after determining if some potential vulnerabilities are discovered, the *engine* simulates the behavior of the agents. For each agent, according to the various parameters previously discussed, the next attack is chosen, unless the agent has already reached its goal or is busy because still implementing the previous attack. To select an attack, a subset of the attack graph is dynamically built, according to the look-ahead value. Then, the selection of the strategy is evaluated.

At the end of a run, the *engine* collects the samples that it stores in a database. Using this database, valuable information is extracted, like the attack chains, the impact, the attacks probability, the reached goals, the number of runs where an agent implements a sequence, and the time to implement a complex attack.

Another very interesting information concerns agent *plans*. A plan is a subsequence of an attack chain without useless attacks. An attack is useless if the agent does not use the rights it grants to reach a goal. Haruspex derives the corresponding agent plans from each sequence through a backward scan that removes useless attacks.

Chapter 3

Framework

We present the main problems posed by the design of a SIEM framework and their solutions. First of all, we describe the general architecture of the proposed SIEM framework and then discuss the details of each module. We also identify the useful information to be derived from the assessment database produced by the Haruspex suite.

3.1 Architecture overview

The proposed framework considers as source devices both kind of IDSs, network and host ones. A variety of widely deployed sensors are available off-the-shelf, and the proposed framework aims to cooperate with them to simplify its adoption in real environments. A proper module, the *Receiver*, collects the alerts raised from the sensors and maps them into a uniform format the other modules can understand.

After being processed by the *Receiver*, the alert has to be validated. The *Filter* module implements the validation process and, eventually, filters out the alert. Furthermore, this process prevents the overload of the SIEM. If the alert is not validated, unknown updates or configuration changes to the target system may have introduced a new vulnerability that may invalidate the previous assessment. Hence, the alert requires further investigations by a security expert.

The *Matcher* and the *Correlator* modules process a validated alert.

The *Matcher* accesses an agent attack patterns database to match the detected sequence of attacks against the agents patterns and identify the attackers.

The *Correlator* correlates the alert with the previous ones. The correlation process basically matches the detected attack sequence against the simulated ones to discover the complex attacks a set of agents are implementing. The correlation process may produce two distinct outputs.

If the correlation is successful, the *Predictor* attributes the detected sequences to some agents and predicts attacks. The attribution of a detected sequence to an agent is a valuable information. First of all, this supports the anticipation of the goals the attacker is trying to achieve. Moreover, we supply some forensics. Obviously, the prediction of the next attacks of an agent is useful to stop its attempt and prevent any impact.

If no correlation is possible because of the mismatch between the sequence and the database, the *Investigator* module analyzes the previous correlation result and the current alert to discover *0-day sequences*. A *0-day sequence* is a complex attack that has not been simulated but that respects the pre and post conditions constraints on the attacks in a sequence. However, in this case we do not know the agent and the *Predictor* cannot predict future attacks. The security expert is informed and, eventually, an update to the Haruspex database may be considered.

Figure 3.1 sketches the whole SIEM architecture.

The following sections describe in more details the SIEM modules. Let us first introduce some hypothesis on sensors.

3.2 Sensors

Each sensor sends alerts to the *Receiver* module trough a secure channel. We assume that the sensors detect all the attacks, both remote and local ones, to the system. This assumes a proper configuration and placement of the sensors.

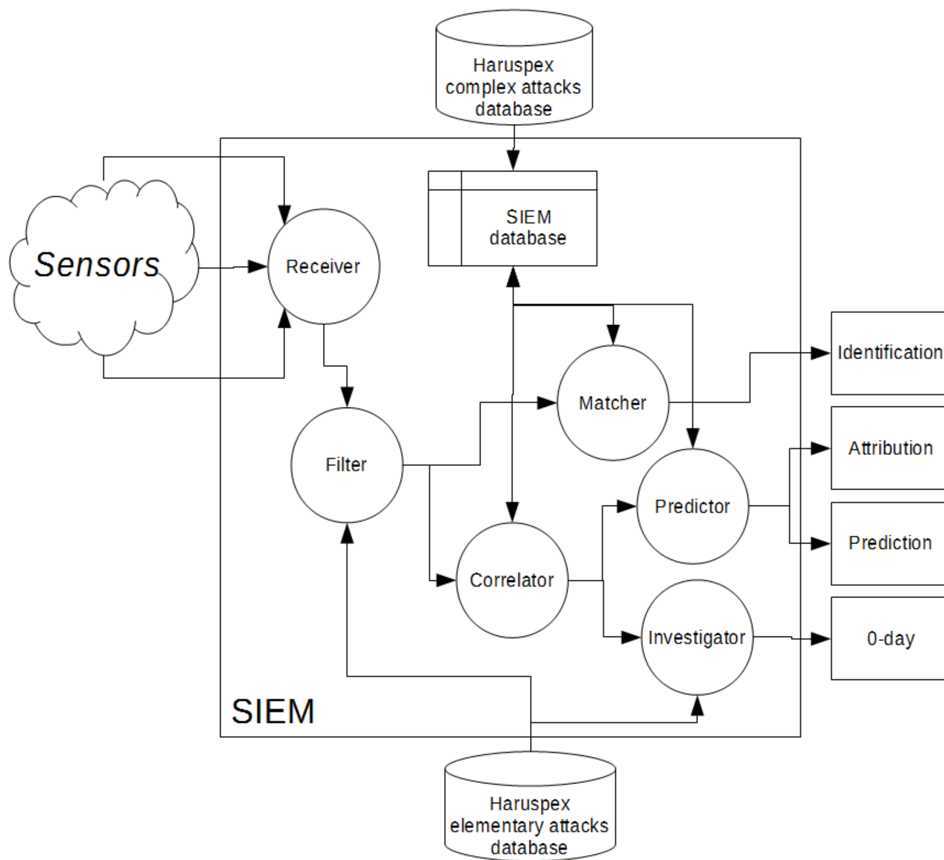

Figure 3.1: The SIEM framework architecture

Furthermore, we assume the sensors do not miss any attack. This implies that no false negatives are possible.

We consider two cases, the *failure detection* and the *attack detection* one. The former assumes that the sensors can distinguish successful attack from failed one. Instead, the latter assumes that the sensors signal attacks without their outcome.

We discuss the main consequences of the two assumptions in the following.

3.3 Failure detection case

This section assumes that the sensors provide alerts for all known attacks together with their success or failure.

3.3.1 Building the SIEM database

First of all, the SIEM has to build a database that is the underlying knowledge and the intelligence of the whole framework. This database is produced from the output of the Haruspex suite, that collects information in multiple simulations of attacks against the system. The suite discovers both the plans and the complex attacks of each agent.

The Haruspex suite maps complex attacks of an agent into the corresponding plans. A plan is a subsequence of an attack chain without useless attacks. Although this seems a useful simplification, by pruning some attacks from a sequence we decrease the differences among the attack chains of distinct agents. This will affect the attribution and prediction phases, because the accuracy of these phases increases with the difference among attack chains. So, agent plans are not optimal for attack attribution and forecasting.

A Haruspex complex attack is a simulated sequence of attacks implemented by an agent and contains both successful and failed attacks. If we denote by at_x^s the successful implementation of attack x and by $at_x^{f,n}$ n con-

secutive failures of attack x, a complex attack may be written as

$$\left[at_1^{f,2}at_1^s at_2^s at_3^{f,4} at_4^{f,3}\dots\right]$$

Because the sensors can detect the result of attacks, the alerts for failed attacks are not interesting because they do not change the security status of the system. So, we can drop failed attacks from a complex one.

By removing failed attacks from a sequence, the SIEM builds a database of *successful pure sequences*. A successful pure sequence is defined as follows.

Definition 1. A *successful pure sequence* is the sequence of elementary attacks extracted from a complex one by removing the failures.

As an example, given the complex attack

$$\left[at_{54}^s at_{63}^{f,4} at_{66}^s at_{23}^{f,2} at_{23}^s at_{63}^s\right]$$

the corresponding successful pure sequence is

$$[at_{54}at_{66}at_{23}at_{63}]$$

We can notice that distinct complex attacks can be mapped into the same successful pure sequence. As a result, the SIEM pairs each successful pure sequence *sps* with the set of agents implementing it and their relative frequencies, that is the number of times the agent implements it in the simulations. If an agent *ag* implements *sps*, we denote its frequency with *freq(sps, ag)*. We can define also *freq(sps)*, that is the number of times *sps* is implemented by any agents in the simulations.

The SIEM stores this information in the SIEM database.

3.3.2 Receiving and filtering alerts

The *Receiver* processes each alert to extract the characteristics of the attack. Distinct sensors can be adopted and each sensor has its own alert format. As a consequence, the *Receiver* needs to parse the alert.

At first, the *Receiver* retrieves the result of the attack. If the alert reports a failed attack, the *Receiver* just logs the corresponding information, without further processing. We can explain this behavior under the assumption that alerts of failed attacks do not convey useful information. This reduces the computational load of the whole SIEM.

We have already seen the attributes to identify an elementary attack. This is also the information the *Receiver* extracts from an alert. These attributes are:

- the target IP
- the target port
- the transport protocol
- the set of references to the vulnerabilities exploited, namely the CVE identifiers

From this perspective, we can define an alert as follows.

Definition 2. An *alert* is the notification of an alarm raised by a sensor that has detected a known attack. Basically, an alert is a tuple of

$$\langle IP, port, protocol, \{CVEids\}\rangle$$

While the first three attributes are always meaningful, the last one, the CVE ids set, may be empty, have a single reference or more than one, because the sensor generating the alert could not determine which vulnerability has been exploited. This could happen because no CVE id has been assigned or because attack vectors are very similar and the sensor could not distinguish among the corresponding vulnerabilities. Consider, for example, the vulnerabilities enabling a SQL Injection attack. Since the query string could match distinct descriptions, the sensor could not be able to identify which vulnerability has been exploited.

Trough this information, the *Filter* finds the Haruspex elementary attacks that are compatible with the alert. The matching condition depends upon

the alert reference set. If this set has at least one CVE id, the *Filter* applies the following definition.

Definition 3. An alert is *compatible* with an elementary attacks if

(i) the IP, port and protocol are the same

(ii) the attack CVE id is contained in the alert reference set of CVE ids

If the alert reference set is empty, the vulnerability exploited is unknown, so the *Filter* cannot apply condition *(ii)*. We consider the worst case scenario, so potentially, any elementary attack, that respect just condition *(i)*, could be exploited. As a result, the *Filter* returns these attacks.

A consequence of the notion of *compatibility* is that the *Filter* could map the alert to a set of Haruspex elementary attacks. This set of attacks may be empty or include more than one attack. An alert mapped into several attacks is a source of non determinism that reduces the accuracy of the next phases.

If an alert is not compatible with any attack, it signals an inconsistency with the system model of the Haruspex suite. As an example, this may be due to an unknown update to the system. Further investigations are required, so the SIEM informs a security expert.

Notice that, since no false positives are possible, if a sensor detects a successful attack then a vulnerability has been exploited.

If at least one attack is compatible with the alert, the *Matcher* and the *Correlator* modules continue the SIEM processing.

3.3.3 Agent attack patterns recognition

The *Matcher* module identifies attacking agents by matching the alert stream against the pattern of the various agents. A *pattern* is defined as follows.

Definition 4. A *pattern* is an ordered sequence of attacks that uniquely identify an agent.

We can see a pattern as a *signature* of an attacker.

First of all, the *Matcher* builds an agent attack pattern database out of the SIEM one.

Starting from a successful pure sequence, the *Matcher* extracts the combinations of N attacks by preserving the sequence order. By iterating this procedure a number of times equal to the length of the extracted sequence, the *Matcher* finds all the combinations for a successful pure sequence. By applying this method to all the successful pure sequences, the *Matcher* constructs for each agent the corresponding set of attack combinations.

Notice that distinct successful pure sequences can generate the same attack combination. So we can pair each of them with its frequency.

Furthermore, the same attack combination could appear in more than one agent set, so the *Matcher* filters out the combinations that are not unique for an agent. The resulting sets of attack combinations are the patterns for the agents. Each pattern is paired with its frequency, the relative attack combination frequency.

The whole procedure has a large time complexity. The *Matcher* provides three mechanisms to cope with this complexity. The user can bound the max value of N and can set frequency thresholds on both successful pure sequences and patterns. Obviously, if any of these mechanisms is used, the *Matcher* neglects some successful pure sequences and patterns. This decreases the accuracy of pattern matching and it may result in false positive or false negative agents identification.

On receiving a new set of compatible attacks from the *Filter*, the *Matcher* searches for totally matched patterns. If at least one is found, the corresponding agent is identified as attacking the system.

We remark that attacking agents are identified through pattern matching and attribution. The two processes are different, even if they aim to achieve the same goal. The *Matcher* analyzes the attack stream, as it is, to recognize known agent attack patterns. Instead, the *Predictor* relies on the correlation process to apply both statistical and heuristic methods to identify the agents. Furthermore, an attribution is paired with a probability value while no probability can be paired with pattern matching.

3.3.4 Correlating attacks

The correlation of alerts is one of the most complex SIEM task. The *Correlator* module implements this functionality by sequentially composing the alerts received according to the attack chains discovered by simulations. The correlation result is the input of the *Predictor* module, that attributes and predicts agent attacks.

Now we give more details on the correlation process.

For simplicity sake, at first we assume that just one agent is attacking the system, but this agent and its goals are unknown. This means that any attack the sensors detect belongs to an attack sequence implemented by that agent.

Informally, the alert stream received up to a given time *matches* a successful pure sequence if each alert of the stream is compatible with the elementary attacks in the successful pure sequence, and it respects their order. This matching could be *complete*, if the condition holds for all the attacks in the successful pure sequence, or *partial*, if the condition holds for the first n attacks of the successful pure sequence. This is a prefix matching between alerts and successful pure sequences.

The correlation result is the set of successful pure sequences that match the alert stream.

On receiving an alert mapped into a set of Haruspex elementary attacks, the *Correlator* removes from the previous correlation result all those successful pure sequences that do not match anymore with the alert stream. This implies that the resulting set could even be empty if no sequence matches with the alert stream. In this case, no agent has implemented the chain in a simulation. This signals an inconsistency, and the *Investigator* module is invoked to distinguish between *0-day sequence* and completely unexpected attack chains.

Now we extend the correlation process to the one where some agents attack the system simultaneously. This is the most general situation that also covers the previous case. So, it is also the default behavior of the SIEM.

If several agents are concurrently attacking the system, the alert stream is

the ordered interleaving of attacks in the sequences of these agents. Since we assume these agents cannot cooperate, each attack in the ordered interleaving is implemented by exactly one agent and the rights it grants are acquired only by this agent. Obviously, the agents and their goals are not known in advance.

Here, each alert of the stream has to be compatible with and in the same order of the ordered interleaving of attacks of a successful pure sequence set.

We can notice that the combinations of distinct successful pure sequences may generate the same ordered interleaving.

Consider, as an example, three successful pure sequences of distinct agents. The first one, sps_1, is $at_1at_3at_5$, the second one, sps_2, is $at_2at_4at_6$, and the third one, sps_3, is $at_1at_2at_3at_4$. If the detected alert stream das is mapped into $at_1at_2at_3$, it is obvious that das could be the ordered interleaving of sps_1 and sps_2, or the single sps_3.

As a consequence, more than one set of successful pure sequences could verify the matching condition. In the previous example, we have two sets. One includes sps_1 and sps_2, while the other one only includes sps_3.

An interesting property is that if an alert mapped into the same elementary attack is received more than one time, it will be matched with at least two different successful pure sequences. We can prove this property by the assumption that a single attack chain never repeats a successful attack.

Moreover, these successful pure sequences can be grouped by the agent that implements them.

The consequence of all these hypotheses is that the correlation process returns a set of tuples, each with two components for each attacking agent. The first component represents the contribution of that agent to the alert stream, while the second is the set of the relative matching successful pure sequences. Because of this interpretation, each of these tuples represents the security status of the system, that is a distinct and alternative *explanation* of the ordered interleaving.

Definition 5. If naa denotes the number of attacking agents, con_i the i-th agent contribution and $comppseq_i$ the matching successful pure sequences set

that have con_i as prefix, an *explanation* is a tuple with the structure

$$\langle < con_1, comppseq_1 >_{ag_1}, \ldots, < con_{naa}, comppseq_{naa} >_{ag_{naa}} \rangle$$

At high level, the correlation process may be seen as the computation of a new set of explanations out of the previous alerts and the received one.

For each explanation, the *Correlator* checks if at least one successful pure sequence of an agent still matches the received alert and, in this case, it builds a new explanation. This explanation is equal to the previous one for all the agents but the considered one. The component of the agent is replaced by the new contribution (the previous one plus this alert) and the new matching successful pure sequence set. Implicitly, the *Correlator* assumes that the agent has implemented the attack that has been detected.

The *Correlator* repeats this procedure for each component of the explanation.

The *Correlator* also handles a special case where the received alert is compatible with an initial attack that is the first attack of a complex one. Here, the *Correlator* determines the set of successful pure sequences that have the first attack compatible with the alert and groups this set by the implementing agent.

For each explanation, the *Correlator* builds a new one. This explanation is equal to the previous one for all the agents but the one that initiates a new sequence. The component of the agent is replaced by the received alert as contribution and the relative successful pure sequence set as matching one. If the agent component was not empty, the *Correlator* stores the previous information as *history* of the explanation. To explain this solution consider that an agent could interrupt a previous attack chain to implement a new one.

As a consequence, the complexity of the correlation process increases with the number of agents attacking the system simultaneously. Fortunately, we can assume that the probability that n agents attack simultaneously the system strongly decrease with n. This makes it possible to bound the number of components, and of concurrent attackers, of explanations. This reduces

the complexity while introducing the possibility of error in the correlation result. Because the *Predictor* and the *Investigator* rely on this explanations set, this could also result in a loss of accuracy of attribution, prediction, and 0-day discovery.

Anyway, by focusing on the sequences an agent actually implements, the proposed approach increases the accuracy of the correlation with respect to approaches that adopt an attack graph to describe all the sequences an agent may implement.

All the new explanations computed by the *Correlator* compose the new explanation set. If it is not empty, this set can be analyzed by the *Predictor* to attribute and predict attacks. Otherwise, the *Investigator* module is invoked to process the previous explanation set and the actual alert in order to discover further information.

3.3.5 Attributing and predicting attacks

The *Predictor* implements both attack attribution and prediction. The first one identifies the agent that is implementing the detected attack chain, while the second one forecasts future attacks of the agents that are currently attacking the system.

As attribution and prediction are probabilistic, they could be affected by the a priori estimate of the probabilities that each modeled agent attacks the systems. This estimate could be supplied by the user as input, but it is not derivable by the Haruspex output database. The attacking probability of each agent is a very critical information, because every statistic is conditioned by the a priori probability that each agent is actually attacking the system. As a consequence, the *Predictor* must be aware if these probabilities are known or not.

The *Predictor* tries to pair each explanation with its probability, that is the likelihood of the security status represented by the explanation.

The probability of each explanation can be computed in two cases only. In the first one the user supplies the a priori estimate of the agents attacking probabilities, in the other one all the explanations have the same set of

attacking agents. In all the other cases, the probability of each explanation cannot be computed, because each one considers distinct agents that could be not actually attacking the system.

In the first case, the user supplies the attacking probabilities of each agent. We denote with *P(ag)* this probability for the agent *ag*.

The *Predictor* queries the SIEM database and retrieves *freq(sps, ag)* and *freq(sps)* for each successful pure sequence *sps* of an explanation *expl*.

The *Predictor* evaluates the probability that an agent *ag* is involved in an explanation *expl* as

$$P(ag, expl) = \frac{\sum_{sps \in expl} freq(sps, ag)}{\sum_{sps \in expl} freq(sps)} \cdot P(ag)$$

From this probability, the *Predictor* computes the relative probability of the explanation *expl* as

$$P_{rel}(expl) = \prod_{ag \in expl} P(ag, expl)$$

Obviously, this probability has to be normalized with those of other explanations, so the *Predictor* computes the real explanation probability as

$$P(expl) = \frac{P_{rel}(expl)}{\sum_{expl} P_{rel}(expl)}$$

The second case is one of the two previously mentioned, namely the one where all the explanations produced by the correlation phase involve the same agents as currently attacking the system. This means that the *Predictor* is sure that all these agents attack the system. As a consequence, it applies the previous reasoning to compute the probability of each explanation by considering the *P(ag)* equal to 1 for any agents.

These probability of each explanation will be used both to attribute and predict attacks. If the *Predictor* cannot compute them, it adopts some heuristics to extract some information.

Attack attribution

We consider at first the solution to the attribution problem.

This solution pairs each agent with the probability that is actually attacking the system given the detected sequence of attacks.

If the *Predictor* has computed *P(expl)*, the attribution probability of an agent is simply the sum of the explanation probability where the agent appears.

$$P_{attr}(ag) = \sum_{ag \in expl} P(expl)$$

Otherwise, the *Predictor* applies a simple heuristic. If an agent ag is present in all explanations, its $P_{attr}(ag)$ is equal to 1, otherwise it is undefined. This is justified by the fact that we are sure that an agent that appears in each explanation is attacking the system, while the *Predictor* cannot say anything about the others. Notice that this can be deduced from the second case to compute of the probability of an explanation previously described.

We define the result of the attribution process in the following way.

Definition 6. If we denote with na the number of agents and with $P_{attr}(ag_i)$ the attribution probability of the agent i computed as previously described, the *attribution* is a tuple with the structure

$$\langle P_{attr}(ag_1), \ldots, P_{attr}(ag_{na}) \rangle$$

Prediction computation

We describe now how a prediction is computed.

In principle, a prediction is a set of couples, each defining an attack and the probability it will detected as the next one. This holds if there is a single attacker, but we are considering a more complex scenario where distinct agents are concurrently attacking the system. As a consequence, the observed alert stream is the ordered interleaving of the attack sequences of some agents. Under these assumptions, the *Predictor* cannot predict in any way who implements the next attack. The *Predictor* can just anticipate the

behavior, so future attacks, of a single agent. Hence, the prediction will be defined as follows.

Definition 7. If we denote with na the number of agents and with $naps_{ag_i}$ the set of next attacks probabilities for the agent i composed by couples of

$$\langle at_x, P_{next}(at_x)\rangle$$

where at_x is an attack x and $P_{next}(at_x)$ is the probability of the attack x to be the next one, the *prediction* is a tuple of

$$\langle naps_{ag_1}, \ldots, naps_{ag_{na}}\rangle$$

Obviously, a prediction strongly depends upon the explanations that the *Correlator* returns, because they describe the last attacks implemented by the agents in their sequences, so the *Predictor* can reason on the next attacks. Because each explanation is alternative to the others, the *Predictor* computes a prediction for each explanation and it may associate each prediction with a probability. This probability is equal to the relative *P(expl)*, if the *Predictor* has been able to compute them, and it is undefined otherwise.

The *Predictor* computes $P_{next}(at_x)$ in the following way.

Each explanation includes, for an agent *ag*, its contribution con_{ag} and its set of matching successful pure sequences $comppseq_{ag}$ that have con_{ag} as prefix. Starting from this information, the *Predictor* determines, for each successful pure sequence *sps* in $comppseq_{ag}$, the last attack at_{last} implemented by *ag* as the last attack of con_{ag} in *sps*. The attack at_x immediately following at_{last} in *sps* will be the next attack of the relative *ag*. Moreover, at_x could be shared among distinct successful pure sequences of the agent. The *Predictor* computes $P_{next}(at_x)$ as a weighted average.

If we denote with $next(sps, at_x)$ the predicate that indicates if at_x is the next attack of *sps*, $P_{next}(at_x)$ is defined as

$$P_{next}(at_x) = \frac{\sum_{next(sps,at_x)} freq(sps, ag)}{\sum_{sps} freq(sps, ag)}$$

By collecting these probabilities for all the next attacks of an agent ag, the *Predictor* computes the $naps_{ag}$ previously described. By applying this procedure for each agent of the explanation, the *Predictor* computes the relative prediction.

Notice that the next attack at_x of a successful pure sequence is the next one that characterizes the privilege escalation an attacker implements to reach its goals. So, the prediction process discovers the next attacks that an attacker has to successfully implement to achieve its goals. It also pairs these attacks with their probability of a future exploitation.

While the attribution returns a single result for all the explanation set, the prediction returns a set of predictions, one for each distinct explanation in the current security status of the system.

In the attribution process, the *P(expl)* support a more accurate estimation of agents attribution probabilities. Anyway, even if *P(expl)* cannot be computed, the *Predictor* can apply heuristics to product results that are accurate and valuable.

The prediction set is always computed but, without *P(expl)*, the *Predictor* cannot compute the most likelihood forecast.

From a concrete security perspective, the *Predictor* produces very interesting results for the real-time prevention of impact. By attributing attacks to an agent we can also discover its goals, so that the SIEM can reduce or even completely avoid the impact by stopping some of the attacks the agent needs to reach a goal.

3.3.6 Discovery of 0-day

When the correlation process cannot return a result, the *Investigator* tries to discover further information on the last alert received. In particular, the *Investigator* can discover a *0-day sequence* out of the last valid correlation result and the actual alert. The whole process is based on the pre and post conditions of elementary attacks.

We first define the *sequence rights* that is the rights set that an agent can

acquire by implementing a sequence of attacks.

Definition 8. A *sequence rights* is the set of rights granted by an attack sequence if all its attacks are successful. If *as[i]* denotes the i-th attack of the attack sequence *as*, *len(as)* the length of the attack sequence *as*, and *post(at)* the postconditions of an attack *at*, the sequence rights *sr(as)* of *as* is the union of all the postconditions, that is

$$sr(as) = \bigcup_{i=0}^{len(as)} \{post(as[i])\}$$

The *Investigator* has to consider all the explanations in the last valid correlation result, because each codifies distinct and alternative security status of the system. Each explanation stores the contribution of each attacker to the alert stream. Under the assumption that the agents do not cooperate, a contribution represents the sequence of successfully attacks by an attacker. In this way, the *Investigator* finds the sequence rights of the contribution to deduce the rights an agent acquires.

Given the right set of the contribution and the initial rights of an agent, the *Investigator* computes the union of these two sets as the *global rights* the agent acquires at the time.

We recall that the actual alert is mapped into a set of Haruspex elementary attacks where each attack has its own preconditions.

Furthermore, a vulnerability has enabled the attack causing the alert and it can be exploited because the attack sequence of an agent grants the proper rights.

The *Investigator* checks if the rights of the agent includes the precondition of a mapped attack. If at least one attack verifies this condition, the alert satisfies the post and pre conditions ordering, so the *Investigator* identifies a possible *0-day sequence* of the agent, namely a sequence the agent has not selected in any simulation.

If, instead, no contribution of an agent to an explanation satisfies the previous condition, the *Investigator* deduces that the attack chain is not consistent with the system model of the Haruspex suite. This could mean

that either the agent has exploited, before this detected attack, a 0-day vulnerability (that cannot be detected by definition), or unknown updates to the target system, e.g. the insertion of a new LAN node, have introduced further vulnerabilities.

The *Investigator* finds other useful information by further investigations.

By considering information on the topology of the target system, the *Investigator* determines the set of nodes where the 0-day vulnerability could have been exploited.

Furthermore, by considering pre and post conditions, the *Investigator* computes the difference between the preconditions of the attacks mapped from the alert and the global rights. This is the set of rights that the 0-day vulnerability could have granted to an agent. Similar reasoning are applied to the sequences rights of the contributions. Here, the *Investigator* identifies the set of rights granted by the 0-day vulnerability with respect to attack chains, rather than to the privilege escalation the agent actually attempts.

As a result, the output of the *Investigator* helps the security experts in the analysis of the unexpected attacks to the system, both for system hardening and vulnerabilities discovery.

3.4 Attack detection

This section highlights the main differences of the framework with respect to the previous case. Now the sensors cannot distinguish successful attacks from failed ones. This is the most general case.

The new SIEM database

Since the sensors cannot determine the result of the attacks, the previous Definition 1 of *successful pure sequence* has to be generalized to take into account the repetition of attacks due to failures.

A complex attack may execute each attack a variable number of times that depends upon an agent attribute as well as on the simulation randomization. We define the general *pure sequence* as follows.

Definition 9. A *pure sequence* is the sequence of elementary attacks extracted from a complex attack, by removing consecutive repetition of the same attacks.

As an example, if the complex attack is

$$\left[at^{s}_{54}at^{f,4}_{63}at^{s}_{66}at^{f,2}_{23}at^{s}_{23}at^{s}_{63}\right]$$

the corresponding pure sequence is

$$[at_{54}at_{63}at_{66}at_{23}at_{63}]$$

As a consequence, the length of a pure sequence is equal to or larger than the one of the successful pure sequence extracted from the same complex attack.

The SIEM database is built in the same way as before, but now the SIEM considers pure sequences rather than successful ones.

Alerts collection and filtering

The *Receiver* still collects alerts from the sensors, but now it cannot retrieve the result of the attacks. As a consequence, any received alert must be parsed and sent to the *Filter* module.

This implies that the whole SIEM has to process a larger number of alerts than in the previous case.

Furthermore, if the *Filter* finds no matching attacks, it could signal an inconsistency, but also a false positive. This may be due to several reasons. A possible example is the one where the attacker lacks of information on the LAN node to attack, so it exploits the wrong vulnerability.

Pattern recognition

The *Matcher* builds its agent pattern database as previously described, but this time it considers pure sequences.

Because pure sequences are longer than successful ones, the complexity of pattern extraction increases, both from the time perspective and from the number of patterns generated.

The larger complexity also affects the matching pattern search.

Attack correlation

The *Correlator* just considers pure sequences instead of successful ones. The whole correlation process does not change even if its complexity increases because of attack repetitions.

Consider, as an example, a pure sequence ps where the attack at_x is the last one matched. If the actual received alert is mapped into a set of attacks that contains at_x, it still matches ps. As a consequence, ps participates at least to the next correlation.

Furthermore, ps could appear in distinct explanations that belong to the new explanations set.

Furthermore, consider the following case. Two pure sequences, ps_1 and ps_2, have the same last matched attack, at_x, and belong to two distinct compatible pure sequence sets of distinct agents. The consecutive repetitions of an alert compatible with at_x will be matched with ps_1, or ps_2, or both. As a consequence, if that alert is received two times, the *Correlator* generates three explanations. Two that represent the doubly matching for, respectively, ps_1 and ps_2, and one that represents one match for ps_1 and one for ps_2. Basically, this results from the combination of the alerts on the pure sequences of distinct agents. Obviously, this reasoning can be extended to a larger number of pure sequences and consecutive alerts.

All these considerations result in an increase in the number of compatible pure sequences and in the one of explanations.

Hence, the complexity of correlation strongly increases. However, the security status that is represented could not change significantly because the last attack of a pure sequence that is matched could be the same one as in the previous correlation.

Attribution and prediction

Attribution and prediction work as previously, because the explanation structure and meaning are not modified. Anyway, the *Predictor* complexity increases because the larger cardinality of the explanation set.

The main difference is in the interpretation of the prediction that now returns the next attacks of the agent and these attacks may even fail. As a consequence, we have a more general view of attack forecast.

0-day discovery

In case of mismatch in the correlation process, the *Investigator* performs further investigation to discover a *0-day sequence*.

Now the contribution represents the sequence of potentially successfully attacks by an attacker, so the *Investigator* has to consider anyway all the attacks of the contribution to compute the sequence rights of Definition 8. This could lead to a set of acquired rights larger than the set of rights the attacker gains, so it could impair the *0-day sequence* discovery capability of the framework.

We can derive a general conclusion from all these differences with the previous case. The lack of information on the result of the attacks affect both the performance and accuracy of the SIEM.

Chapter 4

Implementation

This chapter outlines a prototype implementation of the proposed framework and details its main algorithms and data structures.

4.1 Data structures

This sections details the two main data structures. The first one is the *Pattern Pool* that enables the *Matcher* to match the agent pattern set against the alert stream. The other data structure, the *Pure Sequence Trie*, is used by the *Correlator*, the *Predictor* and the *Investigator*.

4.1.1 Pattern Pool

A pattern is a sequence of attacks that uniquely identify an agent. The *Matcher* matches a set of patterns against the alert stream to identify the corresponding agent. Since the alert stream may be the ordered interleaving of the attacks of distinct agents, the attacks matching a pattern do not have to be consecutive in the stream. In other words, in between each matched attack of a pattern, we may have a number of attacks that are not meaningful for this pattern but match other patterns.

This problem is known in the literature as the *multi-pattern matching with variable length of do not cares.*

Since we have to match each pattern attack by attack, we adopt a simplified version of [26, 50].

The *Pattern Pool* is a data structure that supports the simultaneous matching of a set of patterns. It is an array with one element for each agent. *PP[ag]* denotes the element of the array for agent *ag*. Each element is an hash table indexed by the attack identifier. The value associated with each key *atk* is the set of patterns that have *atk* as the next attack to match. *PP[ag][atk]* denotes this set of patterns.

The *Pattern Pool* represents each pattern as a couple with the structure

$$\langle pat, j \rangle$$

This couple denotes that the pattern *pat* is matched till the j-th attack, and represents the *pattern status*.

Algorithm 4.1 shows the initialization of the *Pattern Pool*.

The *BuildEmptyPatternPool* function allocates in memory the data structure. Then, for each agent element *ag* and for each *atk*, *PP[ag][atk]* is associated with the empty set.

The last loop distributes the patterns in the *PP*. For each pattern *pat* of *ag* in the *agentPatternSet*, the algorithm adds the couple $\langle pat, 0 \rangle$ to the corresponding set *PP[ag][pat[0]]*. Here and in the following, *pat[i]* indicates the i-th+1 attack of the pattern *pat*. So, *pat[0]* is the first attack of *pat*.

At the end, the procedure returns the initialized *PP* with the initial matching status of each pattern.

Algorithm 4.5 shows how the *Matcher* matches patterns against the alert stream.

4.1.2 Pure Sequence Trie

The *Pure Sequence Trie* (PST) is the main in-memory data structure of the current implementation. It is used by the *Correlator*, the *Predictor* and the *Investigator*.

To choose the actual representation of this data structure, first of all we consider that it represents the *agPureSequenceDB*, a database that stores the

Algorithm 4.1 Initialization of the Pattern Pool

Input: $agentPatternSet$, the set of agent patterns
1: $PP \leftarrow BuildEmptyPatternPool()$
2: **for all** ag in $agentSet$ **do**
3: **for all** atk in $attackSet$ **do**
4: $PP[ag][atk] \leftarrow \emptyset$
5: **end for**
6: **end for**
7: **for all** (ag, pat) in $agentPatternSet$ **do**
8: $Add(PP[ag][pat[0]], (pat, 0))$
9: **end for**
10: **return** PP

pure sequences implemented by the agents and their frequency. Each pure sequence is a sequence of attacks and its prefix may be shared with a set of distinct pure sequences. Furthermore, the matching of the alert stream against a set of pure sequences resembles a prefix matching.

These considerations resulted in the selection of an ordered tree or trie data structure, where each trie node represents a pure sequence prefix *p*. So, a node is connected to its children through the set of pure sequences that share *p*. A node includes some pointers that are labeled with the next attack *atk* of *p*. So, each child represents the pure sequence set that share the prefix *p;atk*. This reduces both memory requirements as well as the time to retrieve matching pure sequences.

As far as concerns the correlation process, our main concern is the sequential composition of alert to match a pure sequence set. By using the described trie, this process can be implemented by following a path in the trie. The path is determined according to the sequence of attacks compatible with each alert of the stream. Furthermore, to simplify the computation of an explanation, a node identifies both the contribution and the matching pure sequence set of an agent. As a consequence, each explanation includes pointers to trie nodes. This proves that the trie can efficiently support the correlation.

In its execution, the *Correlator* may need to send pointers to trie nodes to the *Predictor* and the *Investigator*. As a consequence, each node stores

information that also support attribution, prediction, and investigation. In particular, each node stores:

- *children*, the hash table indexed by the next attacks of the prefix. It links the node to its children,
- *prefixFreq*, the frequency of the prefix,
- *agPrefixFreq[ag]*, the frequency of the prefix for each agent *ag*,
- *agNextAtkFreq[ag][atk]*, the frequency of the next attacks *atk* of each agent *ag*,
- *acquiredRightSet*, the right set that an agent that reaches the node acquires.

We will see in the following how the various task use each information.

This data structure is shared among several activities. However, since no activity updates the structure, it cannot become a bottleneck for the parallel computation.

The *Pure Sequence Trie* implements the described trie. Algorithm 4.2 details its construction.

First of all, the function *BuildEmptyPSTrie* allocates the memory for an empty trie.

For each pure sequence *ps*, the algorithm applies a common schema for trie construction. Starting from the root, either it accesses an existing node or creates a new one by following the path determined by each attack *atk* in *ps*.

The function *GetOrAddChild* returns the child of *node* labeled with the attack *atk* if it was already created, otherwise it allocates a new node for *atk* and appends it to the *children* of *node*. Moreover, if the *failureDetection* parameter is set to false, the new node has a self transition to itself, obviously labeled with *atk*. This is useful in the correlation process to take repeated attacks. The function also initializes the *acquiredRightSet* to the union of the *acquiredRightSet* of the parent node and the postconditions granted by *atk*.

Then, the algorithm updates the frequencies with the functions *UpdateNextAtkFreq*, *UpdateAgFreq* and *UpdateFreq*. The frequencies are always increased by a *psFreq* factor, the frequency of the pure sequence. Obviously, the algorithm updates the next attack frequency of the parent, while it increases the prefix frequency of the agent and the one of the child.

Finally, the algorithm returns the pure sequence trie it has built.

Algorithm 4.2 Pure sequence trie construction

Input: $agPureSequenceDB$, the database of pure sequences; $failureDetection$, a boolean value representing if the sensors determine the attack result

1: $psTrie \leftarrow BuildEmptyPSTrie()$
2: **for all** $(ag, ps, psFreq)$ in $agPureSequenceDB$ **do**
3:     $node \leftarrow GetRoot(psTrie)$
4:     $child \leftarrow null$
5:     **for all** atk in ps **do**
6:         $child \leftarrow GetOrAddChild(node, atk, post(atk), failureDetection)$
7:         $UpdateNextAtkFreq(node, ag, atk, psFreq)$
8:         $UpdateAgFreq(child, ag, psFreq)$
9:         $UpdateFreq(child, psFreq)$
10:         $node \leftarrow child$
11:     **end for**
12: **end for**
13: **return** $psTrie$

4.2 Algorithms

This section outlines the main algorithms in the current prototype of the discussed framework.

4.2.1 Pure sequence database construction

Algorithm 4.3 shows that the computation of the pure sequence database is very simple. Each record of this database stores information on the agent *ag* that implements a pure sequence *ps* with a frequency *psFreq*. As a conse-

quence, each record has the following structure

$$\langle ag, ps, psFreq\rangle$$

The algorithm computes this database from the Haruspex complex attack database *complexAttackDB*.

From each complex attack *ca*, the function *GetPureSequence* computes the corresponding pure sequence *ps*. Depending upon the *failureDetection* parameter, the failed attacks may be removed from *ca*. This computes the corresponding successful or generalized pure sequence *ps*.

Given *ps* and the agent *ag* implementing it, the function *AddOrUpdate-Freq* searches the record where *ag* implements *ps* in *agPureSequenceDB*. If it is found, the function increments by one its frequency *psFreq*. Otherwise, it inserts the record $\langle ag, ps, 1\rangle$ into the pure sequence database.

Finally, the algorithm returns the complete *agPureSequenceDB*.

Algorithm 4.3 SIEM database construction

Input: $complexAttackDB$, the database of Haruspex complex attacks; $failureDetection$, a boolean value that represents if the sensors determine the attack result

Output: $agPureSequenceDB$, the database of pure sequences

1: $agPureSequenceDB \leftarrow \emptyset$
2: **for all** (ag, ca) in $complexAttackDB$ **do**
3: $\quad ps \leftarrow GetPureSequence(ca, failureDetection)$
4: $\quad AddOrUpdateFreq(agPureSequenceDB, ag, ps)$
5: **end for**
6: **return** $agPureSequenceDB$

4.2.2 Agent pattern set construction

In the following, we present the Algorithm 4.4 to build the agent pattern set.

The algorithm examines each pure sequence *ps* in the agent pure sequence database *agPureSequenceDB*.

If the frequency *psFreq* of a pure sequence is larger than a threshold *psThreshold*, the function *GetAllAtkCombinations*uses the corresponding pure

sequence *ps* to extract all the attack combinations with a length from 1 to *N*, the upper bound on the length of a combination.

The function *AddOrUpdateFreq* inserts an attack combination *atkComb* for agent *ag* into the *agentPatternSet* with as frequency *psFreq*, if *agentPatternSet* does not include it. Otherwise, the frequency of *atkComb* for *ag* is increased by *psFreq*. This takes into account that a pure sequence *ps* with frequency *psFreq* generates an attack combination *psFreq* times.

Finally, the functions *FilterAtkComb* and *FilterNotUniqueAtkComb* remove from the *agentPatternSet*, respectively, the attack combinations with a frequency lower than *patThreshold* and that are not unique for an agent.

After executing the two functions, the algorithm returns the resulting *agentPatternSet*.

Algorithm 4.4 Agent pattern set construction

Input: $agPureSequenceDB$, the database of pure sequences; N, the max length of a pattern; $psThreshold$, the threshold to apply to pure sequence frequency; $patThreshold$, the threshold to apply to pattern frequency
Output: $agentPatternSet$, the set of agent patterns

```
1: agentPatternSet ← ∅
2: for all (ag, ps, psFreq) in agPureSequenceDB do
3:    if psFreq ⩾ psThreshold then
4:       for all atkComb in GetAllAtkCombinations(ps, N) do
5:          AddOrUpdateFreq(agentPatternSet, ag, atkComb, psFreq)
6:       end for
7:    end if
8: end for
9: FilterAtkComb(agentPatternSet, patThreshold)
10: FilterNotUniqueAtkComb(agentPatternSet)
11: return agentPatternSet
```

The overall complexity depends upon not only the main loop, but also upon the inner one, even if its complexity is bound by the user choice of *N*. Since their iterations are independent, both loops can exploit any available parallelism. The only serialization point may arise because the function *AddOrUpdateFreq* may extract the same attack combination from distinct pure sequences. As a consequence, the *agentPatternSet* has to support concurrent

accesses.

4.2.3 Agent pattern matching

For each alert validated by the *Filter*, the *Matcher* runs the Algorithm 4.5, that updates the pattern status of unidentified agents to discover them.

For each validated alert *al*, *CompAtkSet* returns a non empty set of compatible elementary attacks. Each compatible attack *atk* matches the patterns in *PP[ag][atk]*. The algorithm examines this set to update their status and find totally matched patterns.

For each couple $\langle pat, j \rangle$ in *PP[ag][atk]*, the algorithm considers two cases. If *j+1* is equal to the pattern length *len(pat)*, the pattern is totally matched. Here, the algorithm invokes *MarkAsIdentifiedAndSignal* that stops the computation, marks *ag* as identified and signals it to the user. Otherwise, *pat[j+1]* is the next attack to match, so we add $\langle pat, j + 1 \rangle$ to the corresponding *PP[ag][pat[j+1]]*.

We can notice that the attack *pat[j+1]* could be a compatible attack of the alert. This attack will be considered later to update pattern status. As a consequence, the algorithm delays the insertion to avoid any interference with other *PP[ag][atk]* sets.

When all the attacks in *CompAtkSet(al)* have been processed, the algorithm completes any delayed insertion.

This procedure is repeated for each agent that has not already been identified.

Notice that the algorithm can easily exploit any available parallelism. The most complex computation is the loop to update each pattern status, where each pair is independent from the others. As a consequence, this process can be implemented in parallel.

4.2.4 Correlation

The *Correlator* executes the Algorithm 4.6 to correlate each validated alert with the previous ones.

Algorithm 4.5 Pattern matching algorithm

Input: al, the actual alert
Output: the identified agents, if any

```
1: for all ag not already marked as identified do
2:    for all atk in CompAtkSet(al) do
3:       tmp ← PP[ag][atk]
4:       PP[ag][atk] ← ∅
5:       for all (pat, j) in tmp do
6:          if j + 1 == len(pat) then
7:             MarkAsIdentifiedAndSignal(ag)
8:          else
9:             PostponedAdd(PP[ag][pat[j + 1]], (pat, j + 1))
10:         end if
11:      end for
12:   end for
13:   CommitAdds(ag)
14: end for
```

The *Correlator* sequentially composes the alerts by traversing the *Pure Sequence Trie*. Starting from a node, the *Correlator* reaches a new node by following the links in the *children* hash table. The link to follow is determined by the correspondence between the label of a link and the attack compatible with an alert. Since an alert can be compatible with a set of distinct elementary attacks, the *Correlator* can reach distinct nodes in the trie. This implies that the algorithm has to consider a set of nodes rather than a single one.

The *prevState* is an internal variable of the *Correlator* and represents the set of explanations computed through the previous alert. It is initialized to the set that only includes an empty explanation.

Algorithm 4.6 processes each explanation *expl* of *prevState*. It computation consists of two steps.

The first one considers the case where the received alert *al* is compatible with an initial attack.

The predicate *IsCompatibleWithInitialAtk* checks if *al* is compatible with at least one attack that is initial for a sequence. This can be done by matching the *CompAtkSet(al)* against the children of the trie root. If *al* is com-

patible with an initial attack, the algorithm invokes *GetInitAgNodeSet* that returns a set of couple. Each couple associates an agent *ag* with the set of nodes *initPSTNodeSet* labeled with an attack compatible with *al* and with the children of the trie root. For each couple, the algorithm may build a new explanation *newExpl* equal to *expl*, but where the component of *ag* is replaced by *initPSTNodeSet*. Then, the algorithm inserts this explanation into *newState*. According to the framework, this has to be done only if the number of attacking agents involved in an explanation is not larger than a threshold *maxCA*. Anyway, if *ag* already belongs to *expl*, the building of *newExpl* does not increase the number of attacking agents because we replace its component.

The second step considers each component of an explanation *expl* that refers to an agent *ag*. Here, the algorithm computes the possible extension to an agent sequence. Obviously, this cannot be done for the empty explanation.

For each agent *ag* in an explanation *expl*, the algorithm tries to extend the set of nodes *pstNodeSet* that represents the sequence actually implemented. The algorithm evaluates the predicate *IsExtendableFor* that is true only if the alert *al* is compatible with at least one next attack of *ag* in any node in *pstNodeSet*. If *IsExtendableFor* is true, *GetExtNodeSet* returns a set *newPSTNodeSet* that includes the children nodes of nodes in *pstNodeSet* that are compatible with the alert and implemented by *ag*. Then, the algorithm transmits this information to *BuildNewExpl* to build a new explanation and inserts it into the *newState*. Since it replaces an agent component, the algorithm does not need to check the number of attacking agents in *expl*.

Finally, if *newState* is not empty, it is copied into *prevState* variable and it is returned.

Distinct explanations can be analyzed in parallel because the corresponding computations are independent. Furthermore, also each step can be parallelized. The algorithm can access the trie nodes in parallel because they are not updated. A centralization point arises because of the computation of *newState*. The corresponding implementation has to support the concurrent insertion of explanations by avoiding as much as possible the use of locks.

Algorithm 4.6 Correlation algorithm

Input: al, the received alert
Output: $newState$, the new set of explanations

```
1:  newState ← ∅
2:  for all expl in prevState do
3:    if IsCompatibleWithInitialAtk(al) then
4:      for all (ag, initPSTNodeSet) in GetInitAgNodeSet(al) do
5:        if ag ∈ expl OR AttackingAgents(expl) < maxCA then
6:          newExpl ← BuildNewExpl(expl, ag, initPSTNodeSet)
7:          Add(newExpl, newState)
8:        end if
9:      end for
10:   end if
11:   for all (ag, pstNodeSet) in expl do
12:     if IsExtendableFor(pstNodeSet, al, ag) then
13:       newPSTNodeSet ← GetExtNodeSet(pstNodeSet, al, ag)
14:       newExpl ← BuildNewExpl(expl, ag, newPSTNodeSet)
15:       Add(newExpl, newState)
16:     end if
17:   end for
18: end for
19: if newState is not empty then
20:   prevState ← newState
21: end if
22: return newState
```

4.2.5 Attribution and Prediction

The *Predictor* applies Algorithm 4.7 to attribute and predict attacks.

First of all, *ComputeExplProb* computes the probability of each explanation. According to the framework, this function returns meaningful values if either the user defines *agAtkProb*, the attacking probability of each agent, or if all the explanations in *explSet* involve the same set of agents. Otherwise, *explProb* is undefined.

The computation of the probability of each explanation requires the agent prefix frequency and the total prefix frequency. We recall that an explanation includes a set of couples, where each couple defines an agent *ag* and the set of trie nodes representing the matching sequences actually implemented by *ag*. Each trie node includes both the frequencies of interest in the *agPrefixFreq[ag]* and *prefixFreq* variables. So, this computation is very simple.

After the initialization of *attr*, the attribution is computed. If *explProb* is defined, the attribution for agent *ag* is the sum of the probabilities *explProb[expl]* of the explanations *expl* that involve *ag*. Otherwise, a strategy is applied that is implemented by function *AgentInAll*. This function returns the set of agents involved in each explanation. To implement the adopted strategy, the function *Attribute* pairs each agent in this set with a probability equal to 1, while other agents have undefined attribution probability.

Then, the algorithm computes the prediction. First of all, it initializes the set *predSet* to empty. For each explanation *expl* in the set *explSet*, *ComputePrediction* computes the prediction for *expl* and associates each prediction with the corresponding probability. If the probability is computed it is equal to *explProb[expl]*, undefined otherwise.

Notice that the prediction by *ComputePrediction* uses the information in the nodes associated with *ag*. In particular, it needs the prefix frequency of each agent and the one of the next attack. These values are stored, respectively, in the *agPrefixFreq[ag]* and the *agNextAtkFreq[ag][atk]* variables of each trie node.

The function *ComputeExplProb* can be easily parallelized, because the computation of the probability of each explanation is independent from those

Algorithm 4.7 Attribution and prediction

Input: *explSet*, the set of explanations computed by the *Correlator*
Output: *attr*, the attribution; *predSet*, the set of prediction

1: $explProb \leftarrow ComputeExplProb(explSet, agAtkProb)$
2: $attr \leftarrow InitAttribution()$
3: **if** *explProb* is defined **then**
4:    **for all** *expl* in *explSet* **do**
5:       **for all** *ag* in *expl* **do**
6:          $attr[ag] = attr[ag] + explProb[expl]$
7:       **end for**
8:    **end for**
9: **else**
10:    $agSet \leftarrow AgentInAll(explSet)$
11:    $attr \leftarrow Attribute(agSet)$
12: **end if**
13: $predSet \leftarrow \emptyset$
14: **for all** *expl* in *explSet* **do**
15:    $pred \leftarrow ComputePrediction(expl)$
16:    **if** *explProb* is defined **then**
17:       $Add(predSet, pred, explProb[expl])$
18:    **else**
19:       $Add(predSet, pred, undefined)$
20:    **end if**
21: **end for**
22: **return** *attr* and *predSet*

of the others. Furthermore, attribution and prediction can be executed in parallel.

Attribution can be implemented according to a reduce parallel paradigm. Instead, prediction can be implemented through a map parallel paradigm, because the computation of a prediction is independent from the others. The parallelism of the prediction only requires the support of parallel insertions into *predSet*.

4.2.6 Investigation

Algorithm 4.8 shows the algorithm the *Investigator* executes each time it is invoked. According to the framework, it needs the last valid explanation set *lastState* computed by the *Correlator* and the received alert *al*.

The algorithm examines each explanation *expl* of *lastState*.

For each couple $\langle ag, pstNodeSet\rangle$ in *expl*, *GetGlobalRightSet* computes the alternative global right sets. Each element of this set is a tuple $\langle ag, pstNode, rights\rangle$ that represents that *ag* has implemented the sequence determined by *pstNode* to acquire the rights in *rightSet*. This set is the union of the initial rights of *ag* with those it has acquired through the sequence it has implemented. The set of rights granted by the sequence is the value of *acquiredRightSet* of each node. Hence, we have to merge this set and the one of the initial rights of *ag*.

IsAcceptable determines if at least the precondition of one compatible attacks with the alert *al* is included in *rightSet*. If this is the case, the corresponding agent *ag* and the trie node *pstNode* are inserted into *0daySeqSet*.

Finally, if the *0daySeqSet* is not empty, the algorithm signals the possible *0-day sequences* with the alert *al*. Otherwise, further investigations are required to discover potential 0-day vulnerabilities.

Obviously, the main loop determines the complexity of the algorithm. However, the loop can be parallelized by applying a map paradigm, because each *expl* computation is independent. Again, this requires concurrent insertions into a set.

Algorithm 4.8 Investigation

Input: *lastState*, the last valid explanation set computed by the *Correlator*; *al*, the actual alert

Output: information on 0-day sequences or 0-day vulnerabilities

1: $0daySeqSet \leftarrow \emptyset$
2: **for all** $expl$ in $lastState$ **do**
3:    **for all** $(ag, pstNodeSet)$ in $expl$ **do**
4:       **for all** $(ag, pstNode, rightSet) inGetGlobalRightSet(ag, pstNodeSet)$ **do**
5:          **if** $IsAcceptable(rightSet, al)$ **then**
6:             $Add(0daySeqSet, ag, pstNode)$
7:          **end if**
8:       **end for**
9:    **end for**
10: **end for**
11: **if** $0daySeqSet$ is not empty **then**
12:    $Signal0daySeq(0daySeqSet, al)$
13: **else**
14:    $Investigate0dayVuln()$
15: **end if**

Chapter 5

Experimental results - Failure Detection

This chapter presents the testing environment of the prototype and reports the main experimental results. At first, we evaluate the main SIEM capabilities in the *failure detection* case where the sensors can also return the result of the attacks.

5.1 Testing environment

We implemented a prototype of the Haruspex based SIEM to evaluate its main capabilities.

Since we cannot test the prototype in a real environment, we have also implemented a proper software module to generate an alert stream used as the input of the SIEM. This module, the *AlertGenerator*, simulates the alerts produced from the attack chains of an agent set. The module accesses the Haruspex complex attack database to retrieve agent attack sequences to simulate. If the agent set includes more than one agent, the simulation refers to the case of concurrent attacks. As a consequence, the *AlertGenerator* interleaves in a random way the attack chains of distinct agents to produce a single alert stream. This is the stream produced by a sequence of attacks that is the ordered interleaving of the chains. Since we are in the *failure*

detection case, the module generates alerts only for successful attacks.

We simulate each agent set several times to produce reliable statistics. Each simulation considers the case of a single attacker, two or three concurrent agents. The user can specify both the number of simulations per agent and the number of concurrent agents per simulation. For our purpose, we consider at most three concurrent agents, because a larger number of agents corresponds to a scenario with a neglectable probability.

To be fair, in case of simultaneous attackers we consider all the permutations of the agent set of the Haruspex database. So each agent starts as the i-th agent of the alert stream in an equal number of simulations.

Obviously, we have to monitor the behavior of the SIEM during each simulation. So, some proper modules monitor and collect the statistics of interest to produce the experimental results to evaluate the main SIEM capabilities. These modules are the *GlobalMatchStatistics*, the *GlobalAttrStatistics* and the *GlobalPredStatistics*. They measure, respectively, statistics on the identification of agents through pattern matching, the attribution and the prediction.

Since we do not provide any agent attacking probability, the *Predictor* applies the heuristic already described to attribute attacks and it does not pair each prediction with a probability.

5.1.1 Synthetic database

As input database, we use a synthetic one called *Labyrinth Hard*. In the following, we abbreviate it with simply *labhard*. This database includes 2859 elementary attacks and more than 200000 total attack sequences of 6 agents. All agents have the same initial rights set. This implies that the agents start from the same initial attack set. Furthermore, they try to reach the same goal. As a consequence, differences among agents are only due to distinct selection strategies and knowledge levels on the system, represented by the *lambda* attribute. This implies that this database allows us to evaluate also the characterization of agents through their attributes. It is worth noticing this is a worst case situation where the difference between any two agents is

AgID	AVG	Strategy	Lambda
0	31,498	Smart Subnet First	1
1	12,59	max Increment	1
2	12,928	max Increment	2
3	30,507	max Probability	1
4	31,768	max Probability	2
5	36,745	max Efficiency	2

Table 5.1: Agent general information - failure detection case

very small.

5.2 General information

Table 5.1 reports the general information about the agents. Each record represents:

- *AgID*, the agent identifier,
- *AVG*, the average length of the agent attack sequences,
- *Strategy*, the strategy adopted by the agent,
- *Lambda*, the integer representing the degree of knowledge on the system.

As expected, the agents that adopt a *max Increment* strategy have the shortest average sequence length.

5.3 Pattern matching capability

The *GlobalMatchStatistics* measures distinct aspects to evaluate the identification capability of agents through pattern matching. In particular, Table 5.2 reports for each agent:

- *AVG*, the average distance between the first attack of an agent and the one of its identification,

- *PCT*, the percentage of *AVG* with respect to the average length of the sequences of the agent,
- *WID*, the agent probability of being identified before implementing its first attack,
- *UNID*, the agent probability of not being identified at all.

The *GlobalMatchStatistics* measures these features in case of one, two or three concurrent attackers. The *-1*, *-2* and *-3* in the table header identify these cases.

The *GlobalMatchStatistics* computes the *WID* probability as the ratio between the number of times the *WID* condition holds and the number of agent simulations. The same applies for the *UNID* probability that considers the case where the agent is not identified. Both are errors in the identification, but the *UNID* case is worse than the *WID* one, since the agent attacks and reaches a goal before the SIEM realizes it.

Obviously, in all other cases the agent is correctly identified. So, the *GlobalMatchStatistics* computes the agent *AVG* as the ratio between the sum of the distances and the number of times this case occurs.

Table 5.2 shows some interesting results.

First of all, the probability that the pattern matching does not identify some agent is larger than zero. So, a SIEM that only adopts this mechanism to identify an agent may not identify an attacker. In particular, this happens for agent 1 and 2 that have the higher *UNID* probabilities.

The *UNID-1* probabilities are not null also in case of single attacker per simulation. Since the *Matcher* filters attack combinations the are not unique for an agent, it may happen that no pattern for a specific attack sequence is present.

Agents 1 and 2 have also the higher *PCT* values. The *PCT-1* values for agent 1 and 2 are, respectively, 68% and 57%. So, if these agents attack in isolation, the SIEM on average identifies them when they have implemented more than an half of their attack sequences. The *PCT-2* and *PCT-3* values are even larger, but with lower difference between the values of the two

AgID	AVG-1	PCT-1	WID-1	UNID-1
0	5,293	16,805	0	0,002
1	8,594	68,262	0	0,621
2	7,437	57,526	0	0,42
3	8,368	27,43	0	0,047
4	9,53	29,999	0	0,069
5	9,672	26,323	0	0,009

AgID	AVG-2	PCT-2	WID-2	UNID-2
0	4,402	13,975	0	0
1	17,203	136,646	0	0,572
2	16,053	124,174	0	0,3
3	7,856	25,75	0	0
4	8,266	26,019	0	0,004
5	9,878	26,882	0	0

AgID	AVG-3	PCT-3	WID-3	UNID-3
0	4,669	14,824	0,278	0
1	25,174	199,953	0,002	0,519
2	25,517	197,379	0,006	0,22
3	7,704	25,252	0,231	0
4	7,995	25,168	0,215	0
5	9,654	26,271	0,246	0

Table 5.2: Pattern matching capability - failure detection case

agents. Here, we have to remember that potentially attacks by other agents can occur in between. Anyway, in the worst case scenario, agents 1 and 2 may have already implemented a long sequence of attacks.

Agent 0 adopts a *Smart Subnet First* strategy and has the lower *AVG*, *PCT*, *WID* and *UNID* values, so it is the most easy to be identified.

The other agents instead have *PCT* values of 25% in all cases, so they are soon identified. The *Matcher* identifies them, on average, when the agent has implemented a quarter of its sequence, so the SIEM has the remaining 75% of the chain to discover and stop its attempt.

The *AVG* values and the corresponding *PCT* are almost the same for each agent even if the number of concurrent agents increases. This implies that an agent will be identified through pattern matching on average with *AVG* attacks, independently on the number of agents currently attacking the system.

As a general rule, agents that adopt a *max Increment* strategy have a lower probability of being identified. Furthermore, the identification capability of the pattern matching does not change if two agents only differ because of the *lambda* value. This implies that attack combinations differ because of the agent selection rather than because of the *lambda* attribute.

Furthermore, as the number of concurrent agents increases, the *WID* probability increases as well. As a matter of fact, the ordered interleaving of concurrent agents produces sequences of attacks similar to some patterns of other agents. As a consequence, the *UNID* probability decreases.

The *GlobalMatchStatistics* measures the number of times a condition holds on the agent that has been identified set to evaluate the accuracy of the pattern matching. We consider some conditions:

- *PID*, the final identified agent set is equal to the set of simulated agents,
- *NPID*, it includes all the simulated agents and others,
- *ONID*, it includes at least one simulated agent that is not identified, but at least one that is identified,
- *NOID*, it is disjoint with the set of simulated agents.

ConcNoAg	PID-ratio	NPID-ratio	ONID-ratio	NOID-ratio
1	0,805	0	0	0,195
2	0,001	0,715	0,276	0,008
3	0	0,664	0,336	0

Table 5.3: Pattern matching accuracy - failure detection case

Table 5.3 reports the corresponding probability computed as the ratio between each value and the sum of all values.

As we can see, the most accurate results are produced when just one agent attacks. In all other cases, we have a high value of *NPID-ratio*, so the *Matcher* overestimates the agents currently attacking the system. This shows that, at least in the severe case we have considered, the identification of agent through attack pattern matching does not return reliable and precise results.

5.4 Attribution capability

The attribution is similar to the identification of attacking agents through pattern matching. As a consequence, the *GlobalAttrStatistics* measures and produces the same statistics of the *GlobalMatchStatistics*. This simplifies the comparison of the two approaches. Tables 5.4 and 5.5 report these statistics.

Table 5.4 reports the attribution capability values measured by the *GlobalAttrStatistics*.

First of all, we notice that the *Predictor* never identifies an agent before it is actually attacking the system. Furthermore, all the agents are identified. This implies that the attribution is able to correctly identify each agent.

Agents 1 and 2 are the most difficult to identify, as through pattern matching. Anyway, the worst case happens in the most unlikely scenario where three agents attack concurrently. Furthermore, the values of *AVG* and *PCT* of these agents are much lower than the corresponding values for the pattern matching. The difference in the *PCT* values of these agents is larger than before. As a consequence, here the distinct values of *lambda* affect the attribution capability.

AgID	AVG-1	PCT-1	WID-1	UNID-1
0	3,019	9,584	0	0
1	4,039	32,085	0	0
2	2,144	16,588	0	0
3	4,24	13,899	0	0
4	4,786	15,066	0	0
5	6,069	16,517	0	0

AgID	AVG-2	PCT-2	WID-2	UNID-2
0	6,987	22,182	0	0
1	10,783	85,651	0	0
2	7,15	55,307	0	0
3	10,215	33,483	0	0
4	10,283	32,37	0	0
5	11,637	31,669	0	0

AgID	AVG-3	PCT-3	WID-3	UNID-3
0	11,689	37,109	0	0
1	16,059	127,558	0	0
2	12,806	99,054	0	0
3	14,352	47,044	0	0
4	14,896	46,891	0	0
5	16,289	44,329	0	0

Table 5.4: Attribution capability - failure detection case

ConcNoAg	PID-ratio	NPID-ratio	ONID-ratio	NOID-ratio
1	1	0	0	0
2	1	0	0	0
3	1	0	0	0

Table 5.5: Attribution accuracy - failure detection case

For all other agents the AVG and PCT values are larger than before in all the cases but those with a single attacker. Furthermore, these values increase as the number of concurrent agents increases. This shows that the concurrent attacks of distinct agents impair the attribution capability.

Agents 3 and 4 have the same strategy with different *lambda*, but their AVG and PCT values are not so different. This implies that the *lambda* for the *max Probability* strategy does not characterize the agents for the attribution perspective.

Although simultaneous attacks impair the attribution capability, Table 5.5 shows that in all simulations the *Predictor* correctly and precisely identifies the attacking agents. As a consequence, the attribution is accurate and reliable.

5.5 Prediction capability

For each alert of the stream, the *Predictor* computes a set of predictions. From this set, the *GlobalPredStatistics* extracts the set of next attacks of any agent and stores its cardinality paired with the index of the alert. At the end of the simulations, the *GlobalPredStatistics* computes the average cardinality of the next attack sets for each alert index. This procedure is repeated for a single attacker, two and three concurrent agents. Obviously, a lower average cardinality results in a more accurate prediction.

Figure 5.1 plots the graph of the average cardinality of the next attack sets over the number of alerts. The *SingleAg* curve corresponds to a single attacker, *ConcAg2* and *ConcAg3* correspond to, respectively, two and three concurrent agents.

We can see that the three curves have a descending trend. Obviously,

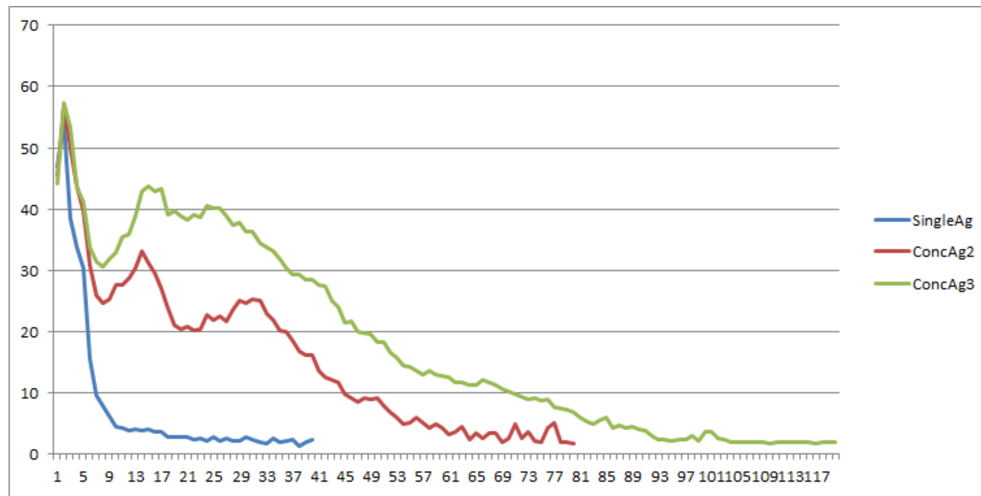

Figure 5.1: The Prediction capability - failure detection case

they have a spike in the first alerts because at the beginning of a sequence an attacker can choose its attack in a larger set.

While the curve associated with a single attacker decreases in a monotonic way, those of the other two cases have some spikes. This may occur when another agent executes its first attack in the simulation.

Since all the curves have a descending trend, the prediction continuously restricts the next attack set of the agents by pruning the set of possible next attacks alert by alert. This happens because the SIEM knows that the sequence of attacks includes only successful ones and we assume that a successful attack is never repeated in a chain. As a consequence, in the *failure detection* case, the prediction can predict future attacks.

Chapter 6

Experimental results - Attack Detection

This chapter reports the same tests previously discussed, but for the *attack detection* case where the sensors signal the detected attack without its outcome.

6.1 Testing environment

Obviously, to produce fair results, the whole testing environment is the same as before. The only difference is that the *AlertGenerator* now generates also alerts for failed attacks. The user can modify the probability of repeating an attack before it is successful.

6.2 General information

Table 6.1 reports the same information on the agents as in the previous section. For each agent, the table shows the average length of attack sequences, strategy and *lambda* in the case of *attack detection*.

Obviously, here the average length of attack sequences is larger than in the *failure detection* case because of the repetition of failed attacks.

AgID	AVG	Strategy	Lambda
0	53,31	Smart Subnet First	1
1	26,496	max Increment	1
2	24,308	max Increment	2
3	42,15	max Probability	1
4	39,728	max Probability	2
5	52,693	max Efficiency	2

Table 6.1: Agent general information - attack detection case

6.3 Pattern matching capability

Table 6.2 reports the pattern matching capability. The AVG values includes also repetition of failed attacks.

As we can see, the considerations done in the previous section for *WID* and *UNID* probabilities still hold. Here there is a little increase of the *WID-2* probabilities, but this could be a statistical fluctuation.

The real difference may be found in the AVG and PCT values. Although the AVG values are larger than the corresponding ones in the *failure detection* case, the values of PCT are lower. This is due to the longer attack chains of the agents. Anyway, larger AVG value also implies that an attacker can even implement a larger number of successful attack before being identified.

Table 6.3 shows an important result. The attack repetition due to failures decreases the *PID-ratio* while it increases the *NPID-ratio* in the single attacker case. This may be explained by considering that attack repetitions can generate the same attack sequence in a pattern of another agent, but, anyway, they also identify the simulated agent. This increases the *NPID-ratio*.

When two or three agents attack concurrently, the values are in line with the corresponding ones of the *failure detection* case.

As a consequence, without the attack result, the pattern matching approach has a lower accuracy.

AgID	AVG-1	PCT-1	WID-1	UNID-1
0	5,572	10,451	0	0,006
1	14,47	54,613	0	0,657
2	12,462	51,266	0	0,407
3	8,85	20,996	0	0,051
4	9,525	23,975	0	0,049
5	12,268	23,283	0	0,008

AgID	AVG-2	PCT-2	WID-2	UNID-2
0	5,201	9,756	0,024	0
1	22,671	85,565	0	0,578
2	23,544	96,854	0	0,278
3	8,501	20,168	0,041	0,002
4	8,986	22,619	0,059	0,002
5	11,114	21,093	0,044	0

AgID	AVG-3	PCT-3	WID-3	UNID-3
0	4,556	8,546	0,287	0
1	33,619	126,886	0,011	0,469
2	34,876	143,471	0,009	0,143
3	9,039	21,445	0,241	0
4	9,743	24,525	0,3	0
5	10,797	20,491	0,269	0

Table 6.2: Pattern matching capability - attack detection case

ConcNoAg	PID-ratio	NPID-ratio	ONID-ratio	NOID-ratio
1	0,404	0,4	0	0,196
2	0	0,728	0,258	0,014
3	0	0,71	0,29	0

Table 6.3: Pattern matching accuracy - attack detection case

AgID	AVG-1	PCT-1	WID-1	UNID-1
0	3,841	7,204	0	0
1	4,268	16,109	0	0
2	3,636	14,958	0	0
3	4,623	10,968	0	0
4	4,797	12,076	0	0
5	7,857	14,912	0	0

AgID	AVG-2	PCT-2	WID-2	UNID-2
0	10,187	19,109	0	0
1	12,769	48,191	0	0
2	10,357	42,608	0	0
3	11,235	26,655	0	0
4	12,367	31,128	0	0
5	14,798	28,084	0	0

AgID	AVG-3	PCT-3	WID-3	UNID-3
0	14,337	26,894	0	0
1	19,985	75,428	0	0
2	16,989	69,889	0	0
3	19,109	45,336	0	0
4	18,385	46,278	0	0
5	21,715	41,21	0	0

Table 6.4: Attribution capability - attack detection case

6.4 Attribution capability

The attribution capability is not affected by the uncertainty on the attack result, as Table 6.4 shows.

Again, all the *WID* and *UNID* values are equal to zero, so there is no error in the attribution.

Obviously, the *AVG* values increase with respect to the *failure detection* case. This is due to several factors. One of them is the repetition of failures. Anyway, also the average agent sequence length increases and this can contribute to improve the *PCT* values.

Notice that now the *Predictor* attributes the Agents 1 and 2 with *PCT* of, respectively, 75% and 69% in the worst case of three concurrent agents. These

ConcNoAg	PID-ratio	NPID-ratio	ONID-ratio	NOID-ratio
1	1	0	0	0
2	1	0	0	0
3	1	0	0	0

Table 6.5: Attribution accuracy - attack detection case

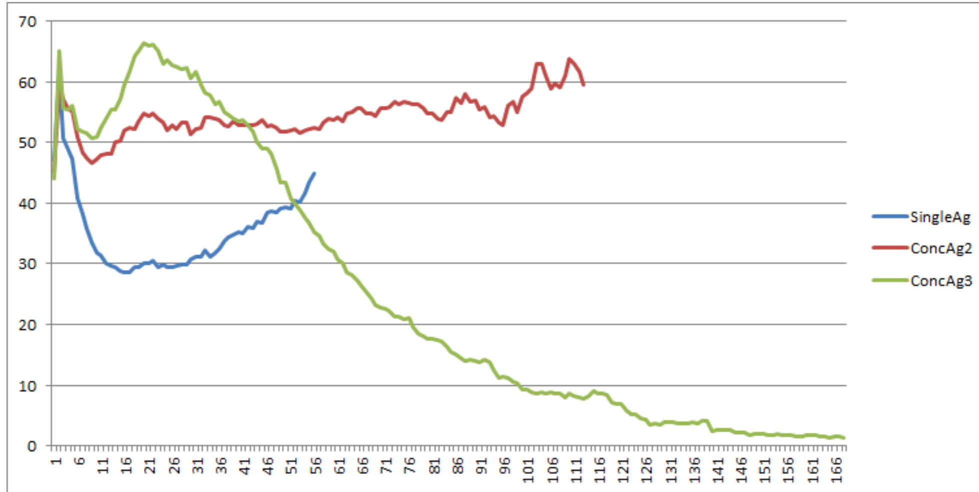

Figure 6.1: The Prediction capability - attack detection case

values are significantly better than the corresponding ones in the previous section. In this case, failed attacks simplify the solution.

Table 6.5 shows that also in the *attack detection* case the attribution returns accurate results.

6.5 Prediction capability

Figure 6.1 plots the average cardinality of the next attack sets with respect to the number of alerts, as in the *failure detection* case.

When a single agent attacks, at first, the curve decreases but then it increases. This implies that, after some attacks, there is an increase in the number of attacks the agents can select to reach their goals.

The curve of two concurrent agents is almost parallel to the X axis but with an increasing trend. This means that the cardinality of the next attack

set increases because we consider both successful and failed attacks some agents can implement.

Only the curve associated with three concurrent agents respects the decreasing trend after a spike.

As a result, since in the *attack detection* case the SIEM cannot known the result of the attacks, the prediction cannot properly reduce the next attack set. This strongly decreases the accuracy of the prediction.

Chapter 7

Conclusions

This chapter summarizes this thesis and its results. Then, it outlines some future works.

7.1 Final remarks

The increasing complexity of the infrastructure of ICT systems strongly increases the complexity of security monitoring because security tools have to rebuild the security status of the whole system to detect and prevent sequences of attacks. This problem is faced in a modular way. A network of sensors detects single attacks, while the *Security Information and Event Management* (SIEM) analyzes these alerts to discover how they are correlated. This correlation exploits a proper knowledge base.

We have defined and implemented a new SIEM framework based on the output of Haruspex, a suite to automate the assessment and the management of ICT risk. The suite simulates the behavior of a set of intelligent agents that may attack the system and it returns a database of the attack sequences these agents can implement and the probability of each sequence. This is the knowledge base that the correlation uses.

Since distinct sensors may be characterized by widely different capabilities, our evaluation has considered two main cases. In the first one, *failure detection*, the sensors can detect not only the occurrence of attacks, but also

their result. In other one, *attack detection*, the sensors can only signal attacks without their outcome.

We have evaluated the capabilities of the prototype of the proposed SIEM through simulations that have considered distinct implementation strategies. In these simulations we have collected several measures to evaluate the reliability and accuracy of the proposed framework. The outcomes of these simulations have shown several interesting results.

The pattern matching approach cannot produce accurate results. It produces the most accurate results only when a single agent attacks the system in the *failure detection* case. In the other cases, it overestimates the set of agents actually attacking the system. As a consequence, the identification of agents through pattern matching is not reliable in practice.

Obviously, any increase in the number of agents that are concurrently attacking the system increases the complexity of the correlation. As a consequence, also the complexity of attribution increases. Even if, on average, the attribution requires more attacks to identify an agent than the pattern matching approach, it is always accurate.

From the pattern matching and attribution perspectives, the selection strategy of an agent is the only attribute that really characterizes it. Instead, its *lambda* attribute affects the SIEM identification capabilities just in one case. This may be explained by considering that while the *lambda* can affect the order of some attacks, the attacks are mostly related to a strategy and a goal, so all the agents sharing these two attributes will implement the same attacks.

Finally, the prediction of future attacks is effective and accurate only in the *failure detection* case. When the sensors can detect attacks but not their outcome, the prediction cannot properly prune the set of next attacks because an agent can repeat the failed ones. As a consequence, the uncertainty on the attack result impairs the prediction.

7.2 Future works

This section outlines some future works on further evaluations and extensions of the framework.

7.2.1 Real environment test

Till now, we have evaluated the prototype though simulations. As a future work, we can deploy the SIEM in a real environment.

Since a network of sensors can produce a high volume of alerts, the SIEM should be able to properly handle this volume. In particular, we can evaluate some aspects not considered in our current experiments.

First of all, we can measure the SIEM capability in validating alerts. This allow us to evaluate the false positive and negative rates that the SIEM recognizes.

By deploying the SIEM in a dedicated machine, we can measure its processing overhead and the throughput it can achieve. The SIEM throughput is a very important measure, because a high throughput implies that the SIEM can process a large number of alerts in a short period. As a consequence, the SIEM can detect and react to complex attacks in a short time.

Furthermore, by changing the configuration of the network of sensors, i.e. their number and the subset of the system that is monitored, we can also evaluate the throughput each configuration requires as a function of the number and kind of sensors.

Finally, we can evaluate the SIEM capabilities by deploying a distributed version of the framework.

7.2.2 Sensors deployment and ruleset generation

The framework has assumed a proper placement and configuration of the sensors. If we relax this assumption, the SIEM has to produce information on both these aspects.

The SIEM now includes also a proper module that receives as input the database of elementary and complex attacks of Haruspex, the topology of

the system, a database of available sensors, some constraints on them, e.g. the maximum number, and a database that pairs each vulnerability with the corresponding signature.

By analyzing this information, the module can generate the list of sensors to be deployed and the allocation of each sensor. This list pairs each sensor with the rules enabling the detection of an attack set and the optimal placement in the system. This new feature has to be evaluated.

Furthermore, the sensors may not be able to detect some attacks because of their capabilities or placement. As a consequence, we can evaluate the SIEM capabilities in this case.

7.2.3 False positive and false negative handling

Actually, the framework has not considered the case where the SIEM has processed a false positive or a false negative.

The former implies a mismatch in the correlation that may be resolved by removing an attack from the detected sequence. Furthermore, the resulting sequence may match one the agents execute with a large probability. This is a strong indication that the removed attack corresponds to a false positive.

Similar considerations apply to false negative, but the handling has to add attacks to sequences instead of removing them.

Both these solutions require some backtracking capability in the correlation algorithm that may impair the attribution and prediction.

Further evaluations on these aspects are required to understand in more details the correlation capability and its effects on the attribution and prediction.

Bibliography

[1] Seyed Hossein Ahmadinejad and Saeed Jalili: *Alert correlation using correlation probability estimation and time windows.* In *Computer Technology and Development, 2009. ICCTD'09. International Conference on*, volume 2, pages 170–175. IEEE, 2009.

[2] Seyed Hossein Ahmadinejad, Saeed Jalili, and Mahdi Abadi: *A hybrid model for correlating alerts of known and unknown attack scenarios and updating attack graphs.* Computer Networks, 55(9):2221–2240, 2011.

[3] Faeiz Alserhani: *A framework for multi-stage attack detection.* In *Electronics, Communications and Photonics Conference (SIECPC), 2013 Saudi International*, pages 1–6. IEEE, 2013.

[4] Faeiz Alserhani, Monis Akhlaq, Irfan Ullah Awan, Andrea J Cullen, and Pravin Mirchandani: *Mars: multi-stage attack recognition system.* In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 753–759. IEEE, 2010.

[5] James P Anderson: *Computer security threat monitoring and surveillance.* Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, 1980.

[6] Rebecca Bace and Peter Mell: *Nist special publication on intrusion detection systems.* Technical report, DTIC Document, 2001.

[7] Fabrizio Baiardi, Fabio Corò, Federico Tonelli, Alessandro Bertolini, Roberto Bertolotti, and Daniela Pestonesi: *Assessing and managing ict risk with partial information.* 2014.

[8] Fabrizio Baiardi, Fabio Corò, Federico Tonelli, Luca Guidi, and Daniele Sgandurra: *Simulating attack plans against ict infrastructures.* In *Vulnerability, Uncertainty, and Risk@ sQuantification, Mitigation, and Management*, pages 627–637. ASCE.

[9] Fabrizio Baiardi, Fabio Corò, Federico Tonelli, and Daniele Sgandurra: *Automating the assessment of ict risk.* Journal of Information Security and Applications, 19(3):182–193, 2014.

[10] Fabrizio Baiardi, Fabio Corò, Federico Tonelli, and Daniele Sgandurra: *A scenario method to automatically assess ict risk.* In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 544–551. IEEE, 2014.

[11] Leau Yu Beng, Sureswaran Ramadass, Selvakumar Manickam, and Tan Soo Fun: *A comparative study of alert correlations for intrusion detection.* In *Advanced Computer Science Applications and Technologies (ACSAT), 2013 International Conference on*, pages 85–88. IEEE, 2013.

[12] Rory Bray, Daniel Cid, and Andrew Hay: *OSSEC host-based intrusion detection guide.* Syngress, 2008.

[13] Steven Cheung, Ulf Lindqvist, and Martin W Fong: *Modeling multistep cyber attacks for scenario recognition.* In *DARPA information survivability conference and exposition, 2003. Proceedings*, volume 1, pages 284–292. IEEE, 2003.

[14] Tobias Chyssler, Simin Nadjm-Tehrani, Stefan Burschka, and Kalle Burbeck: *Alarm reduction and correlation in defence of ip networks.* In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004. 13th IEEE International Workshops on*, pages 229–234. IEEE, 2004.

[15] Frédéric Cuppens: *Managing alerts in a multi-intrusion detection environment.* In *Computer Security Applications Conference, Annual*, pages 0022–0022. IEEE Computer Society, 2001.

[16] Herve Debar, David A Curry, and Benjamin S Feinstein: *The intrusion detection message exchange format (idmef).* 2007.

[17] Hervé Debar, Marc Dacier, and Andreas Wespi: *Towards a taxonomy of intrusion-detection systems.* Computer Networks, 31(8):805–822, 1999.

[18] Dorothy E Denning: *An intrusion-detection model.* Software Engineering, IEEE Transactions on, (2):222–232, 1987.

[19] Robert D Gardner and David A Harle: *Methods and systems for alarm correlation.* In *Global Telecommunications Conference, 1996. GLOBECOM'96.'Communications: The Key to Global Prosperity*, volume 1, pages 136–140. IEEE, 1996.

[20] Mohammad GhasemiGol and Abbas Ghaemi-Bafghi: *A new alert correlation framework based on entropy.* In *Computer and Knowledge Engineering (ICCKE), 2013 3th International eConference on*, pages 184–189. IEEE, 2013.

[21] Gabriel Jakobson and Mark Weissman: *Alarm correlation.* Network, IEEE, 7(6):52–59, 1993.

[22] Klaus Julisch: *Clustering intrusion detection alarms to support root cause analysis.* ACM Transactions on Information and System Security (TISSEC), 6(4):443–471, 2003.

[23] Dominique Karg and Julio Casal: *Ossim: Open source security information management.* Technical report. `https://www.alienvault.com/open-threat-exchange/projects`.

[24] Rajeshwar Katipally, Wade Gasior, Xiaohui Cui, and Li Yang: *Multistage attack detection system for network administrators using data mining.* In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 51. ACM, 2010.

[25] Igor Kotenko and Andrey Chechulin: *Common framework for attack modeling and security evaluation in siem systems.* In *Green Computing*

and Communications (GreenCom), 2012 IEEE International Conference on, pages 94–101. IEEE, 2012.

[26] Gregory Kucherov and Michaël Rusinowitch: *Matching a set of strings with variable length don't cares.* Theoretical Computer Science, 178(1):129–154, 1997.

[27] Hung Jen Liao, Chun Hung Richard Lin, Ying Chih Lin, and Kuang Yuan Tung: *Intrusion detection system: A comprehensive review.* Journal of Network and Computer Applications, 36(1):16–24, 2013.

[28] David Miller and Brock Pearson: *Security information and event management (SIEM) implementation.* McGraw-Hill, 2011.

[29] MITRE: *Cve, a dictionary of publicly known information security vulnerabilities and exposures.* Technical report. `http://cve.mitre.org/`.

[30] Peng Ning, Yun Cui, Douglas S Reeves, and Dingbang Xu: *Techniques and tools for analyzing intrusion alerts.* ACM Transactions on Information and System Security (TISSEC), 7(2):274–318, 2004.

[31] Stephen Northcutt, Jay Beale, Andrew R Baker, Joel Esler, and Toby Kohlenberg: *Snort: IDS and IPS toolkit.* Syngress Press, 2007.

[32] Serkan Ozkan: *Cve details: The ultimate security vulnerability datasource.* Technical report. `http://www.cvedetails.com/`.

[33] Fabien Pouget and Marc Dacier: *Alert correlation: Review of the state of the art.* Technical Report EURECOM+1271, Eurecom, December 2003. `http://www.eurecom.fr/publication/1271`.

[34] Sebastian Roschke, Feng Cheng, and Christoph Meinel: *High-quality attack graph-based ids correlation.* Logic Journal of IGPL, 21(4):571–591, 2013.

[35] Reza Sadoddin and Ali Ghorbani: *Alert correlation survey: framework and techniques.* In *Proceedings of the 2006 International Conference on*

Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services, page 37. ACM, 2006.

[36] Saeed Salah, Gabriel Maciá-Fernández, and Jesús E Díaz-Verdejo: *A model-based survey of alert correlation techniques.* Computer Networks, 57(5):1289–1317, 2013.

[37] Karen Scarfone and Peter Mell: *Guide to intrusion detection and prevention systems (idps).* NIST special publication, 800(2007):94, 2007.

[38] Mike Schiffman: *Common vulnerability scoring system (cvss).* Technical report. `https://www.first.org/cvss`.

[39] Seongjun Shin, Seungmin Lee, Hyunwoo Kim, and Sehun Kim: *Advanced probabilistic approach for network intrusion forecasting and detection.* Expert Systems with Applications, 40(1):315–322, 2013.

[40] Steven R Snapp, James Brentano, Gihan V Dias, Terrance L Goan, L Todd Heberlein, Che Lin Ho, Karl N Levitt, Biswanath Mukherjee, Stephen E Smaha, Tim Grance, *et al.*: *Dids (distributed intrusion detection system)-motivation, architecture, and an early prototype.* In *Proceedings of the 14th national computer security conference*, volume 1, pages 167–176. Citeseer, 1991.

[41] Stuart Staniford-Chen, Brian Tung, Dan Schnackenberg, *et al.*: *The common intrusion detection framework (cidf).* In *Proceedings of the information survivability workshop*, 1998.

[42] Alfonso Valdes and Keith Skinner: *Probabilistic alert correlation.* In *Recent Advances in Intrusion Detection*, pages 54–68. Springer, 2001.

[43] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A Kemmerer: *Comprehensive approach to intrusion detection alert correlation.* Dependable and Secure Computing, IEEE Transactions on, 1(3):146–169, 2004.

[44] Lingyu Wang, Anyi Liu, and Sushil Jajodia: *An efficient and unified approach to correlating, hypothesizing, and predicting intrusion alerts.* In *Computer Security–ESORICS 2005*, pages 247–266. Springer, 2005.

[45] Lingyu Wang, Anyi Liu, and Sushil Jajodia: *Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts.* Computer communications, 29(15):2917–2933, 2006.

[46] Michael Whitman and Herbert Mattord: *Principles of information security.* Cengage Learning, 2011.

[47] Jinqiao Yu, YV Ramana Reddy, Sentil Selliah, Srinivas Kankanahalli, Sumitra Reddy, and Vijayanand Bharadwaj: *Trinetr: an intrusion detection alert management systems.* In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004. 13th IEEE International Workshops on*, pages 235–240. IEEE, 2004.

[48] Xin Zan, Feng Gao, Jiuqiang Han, and Yu Sun: *A hidden markov model based framework for tracking and predicting of attack intention.* In *Multimedia Information Networking and Security, 2009. MINES'09. International Conference on*, volume 2, pages 498–501. IEEE, 2009.

[49] Tianning Zang, Xiaochun Yun, and Yongzheng Zhang: *A survey of alert fusion techniques for security incident.* In *Web-Age Information Management, 2008. WAIM'08. The Ninth International Conference on*, pages 475–481. IEEE, 2008.

[50] Meng Zhang, Yi Zhang, and Liang Hu: *A faster algorithm for matching a set of patterns with variable length don't cares.* Information Processing Letters, 110(6):216–220, 2010.

[51] Urko Zurutuza and Roberto Uribeetxeberria: *Intrusion detection alarm correlation: a survey.* In *Proceedings of the IADAT International Conference on Telecommunications and Computer Networks*, pages 1–3, 2004.