Mestrado em Engenharia Informática
Dissertação
Relatório Final

A Distributed Security Event Correlation Platform for SCADA

Pedro Guedes Alves
pgalves@student.dei.uc.pt

Orientador:
Professor Doutor Tiago Santos Cruz
Data: 01 de Julho de 2014

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

A Distributed Security Event Correlation Platform for SCADA

by

Pedro Guedes Alves

Critical Infrastructures rely on Industrial Control Systems (ICS) such as Supervisory Control and Data Acquisition (SCADA) to operate the networks and systems of vital assets for the functioning of society and economy. SCADA systems were traditionally isolated and used closed architectures with proprietary protocols, but nowadays this systems use open standards with open architectures that are highly interconnected with other corporate networks and the internet. As a result, the vulnerability of these systems to cyber-attacks increased considerably.

This thesis is integrated in the work developed by the Laboratory of Communications and Telematics for CockpiCI, an European Framework FP7 research project, whose goal is to provide intrusion detection, analysis and protection techniques to Critical Infrastructures.

The design and implementation of an event correlation platform for detection of cyber-attacks in SCADA systems are detailed in this thesis. The developed correlation platform implements the means to collect, process and correlate security events from differently distributed sources. The validation performed to this system demonstrated its resiliency, performance and correlation capabilities to detect cyber-attacks.

The platform presented will be deployed in a test bed that includes critical infrastructures simulated by real equipment and enterprise Industrial Control Systems, this will allow a further validation of its concepts and capabilities.

Keywords: "Event Correlation" "Event Processing" "Distributed Event Correlation" "Intrusion Detection" "SCADA"

Contents

List of Figures

List of Tables

List of Acronyms

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
APL	Apache Software License
BRMS	Business Rule Management System
CEP	Complex event Processing
CI	Critical Infrastructure
DBMS	Database Management Systems
DDS	Data Distribution Service
DOM	Document Object Model
EPL	Event Processing Language
GPL	General Public License
HIDS	Host-based Intrusion Detection System
HMI	Human-Machine Interaction
HTB	Hybrid Test Bed
ICS	Industrial Control System
IDMEF	Intrusion Detection Message Exchange Format
IDXP	Intrusion Detection Exchange Protocol
IPS	Intrusion Prevention System
JCP	Java Community Process
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LCT	Laboratory of Communications and Telematics
MOM	Message Oriented Middleware
MQTT	Message Queue Telemetry Transport
NIDS	Network Intrusion Detection System
OASIS	Organization for the Advancement of Structured Information Standards
OCSVM	One Class Support Vector Machines
p2p	point-to-point
PIDS	Perimeter Intrusion Detection System

List of Acronyms (cont.)

POJO	Plain Old Java Objects
pub/sub	publish-and-subscribe
RTU	Remote terminal Unit
SASL	Simple Authentication and Security Layer
SCADA	Supervisory Control and Data Acquisition
SDEE	Security Device Event Exchange
SEM	Security Event Management
SIEM	Security Information and Event Management
SIM	Security Information Management
SMP	Security Management Platform
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
SQL	Structured Query Language
STOMP	Simple Text Oriented Messaging Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLS	Transport Layer Security
UDP	User Datagram Protocol
XSD	XML Schema Definition

Chapter 1

Introduction

1.1 Motivation and Background

Critical Infrastructures (CIs) rely on Industrial Control Systems (ICS) such as Supervisory Control and Data Acquisition (SCADA) to operate network and systems designed to support industrial processes. These infrastructures encompass vital assets for the functioning of society and economy. Usually include facilities for the generation, transmission and distribution of electricity, facilities for the production, transport and distribution of oil and gas, water supply and sewer processing, telecommunication networks, transportation systems, among others.

SCADA systems are the largest subgroup of among ICS and are used to control assets using a centralized data acquisition and supervisory control. Legacy control systems were isolated and used closed architectures with proprietary protocols, in which the degree of security was achieved by obscurity and system isolation. Nowadays, these systems use open standards with open architectures and ICS systems are highly interconnected with other corporate network and the internet. As a result, the vulnerability of these systems to cyber-attacks increased considerably.

With the full recognition of the risks linked cyber-attacks to CIs several European Framework (FP7) research projects were initiated. CockpitCI is one of those research projects. The goal of this project is to provide intrusion detection, analysis and protection techniques to Critical Infrastructures. CockpitCI will aim to define and implement risk modeling and prediction tools, to collect and share information among different infrastructures and intrusion detection systems to detect attacks and anomalies. The CockpitCI focus is on detecting ongoing attacks and minimizing their impact instead of focusing in avoiding the attack.

As part of this project, the Laboratory of Communications and Telematics (LCT) group is responsible for the coordination of the research and design of the components for the Analysis and Detection Layer of the CockpitCI platform. More specifically, one

of the tasks is to design, develop and integrate a distributed Intrusion Detection System (IDS). This real-time distributed Perimeter Intrusion Detection System (PIDS) should be able to aggregate, filter and analyze information of potential cyber-attacks against SCADA systems used to run the CIs.

This thesis integrates with the work developed within the group and is focused in the design, development and integration of a correlation platform capable of collecting data from local detection agents and correlate this data. The work also includes the design and implementation of the communication infrastructure that allows the collected data to be transmitted between the different components.

1.2 Research Objectives

The aim of this research is to explore the use of event correlation for detection of cyber-attacks in SCADA systems. The research should result in the development of a distributed security event correlation platform suited for industrial control networks. This platform must fulfill the requirements of CockpitCI's Dynamic Perimeter Intrusion Detection System (PIDS).

The ultimate goal of the developed system is to provide the necessary means of collecting, processing and correlating security events from differently distributed sources. As a result, the correlation platform must detect intrusions in industrial control networks as soon as possible so that is possible to minimize the effects of the attack.

The proposed solution shall use existing open source software whenever a proven solution already exists for the core problems to solve. The idea is to streamline the development and allow focus on the innovative aspects. A research of existing open source software related to the scope of the project should allow to identify the software that can be used to build the platform.

To verify and validate the design and implementation, attack scenarios and validation tests are implemented and deployed.

Other components of the PIDS and CockpitCI should integrate with the platform. The integration with the other components of CockpitCI should be carried out in a testbed that realistically simulates a real world SCADA environment.

1.3 Document Outline

In this section is going to be detailed the structure of next chapters in this document.

Chapter 2 presents an overview of the CockpitCI project.

Chapter 3 starts by presenting an overview of the diverse techniques used in event correlation and how they used in the field of security, more precisely in intrusion detection. In Section 3.4 is presented a survey of the existing open source correlation software, where the main characteristics of each one of the tools is described.

In Chapter 4 are described the transport protocols for event communication in distributed systems.

Chapter 5 discusses the correlation platform architecture and components, as well as how this platform integrates in the CockpitCI Perimeter Intrusion Detection System.

In Chapter 6 is described how the correlation platform was implemented and the several components integrated.

Chapter 7 describes the validation performed to the correlation platform and the results obtained.

In the Chapter 8 is given an overview of progress of the work developed for this thesis.

Chapter 2

CockpitCI Project

This chapter gives an overview of the CockpitCI project. It provides better understanding of how the work developed in this thesis is integrated within the scope of the CockpitCI.

2.1 Overview and Objectives

The CockpitCI project continues the work done in the FP7 MICIE[1] project (a Tool for systemic risk analysis and secure mediation of data exchanged across linked CI information infrastructures). MICIE target was to develop a secure online software architecture that shared information on a real time basis among local risk predictors, in order to obtain accurate and synchronized predictions using interdependency models. With this tool, the CI operators receive information about the future evolution of their infrastructure with a wider perspective compared to predictions that can be generated by sector specific and isolated simulators. While the MICIE project has proved that increasing cooperation among infrastructures owners by sharing information leads to better predictions, such integration is not enough in order to quickly and efficiently react to cyber attacks. The CockpitCI project as two main objectives. The first is to improve the MICIE on-line Risk Predictor deployed in SCADA center that shares real-time information among CI owners. The other is to add cyber detection capabilities in order to get a broader perspective in terms of security, to identify in near real time the CI functionalities impacted by cyber-attacks and assess degradation of CI delivery services. CockpitCI aims to classify the associated risk level and activate a strategy of containment of possible consequences of cyber attacks together with the provision of some intelligence to field equipment, allowing it performing local decisions in order to self-identify and self-react to abnormal situations induced by cyber attacks. The main

[1] http://www.micie.eu/

components of CockpitCI are a set of modules, each with a different set of functionalities, as described below:

Cyber Analysis and Detection Layer (DL): provides cyber attack detection and identifies the type of cyber threats;

Integrated Risk Prediction (IRP): provides situation awareness that reflects the current situation, predicts the near-term evolution of the situation with risk prediction and provides reaction by selecting the appropriate countermeasures as well as triggers automatic reaction logic;

Secure Mediation Gateway (SMN): provides secure data exchange;

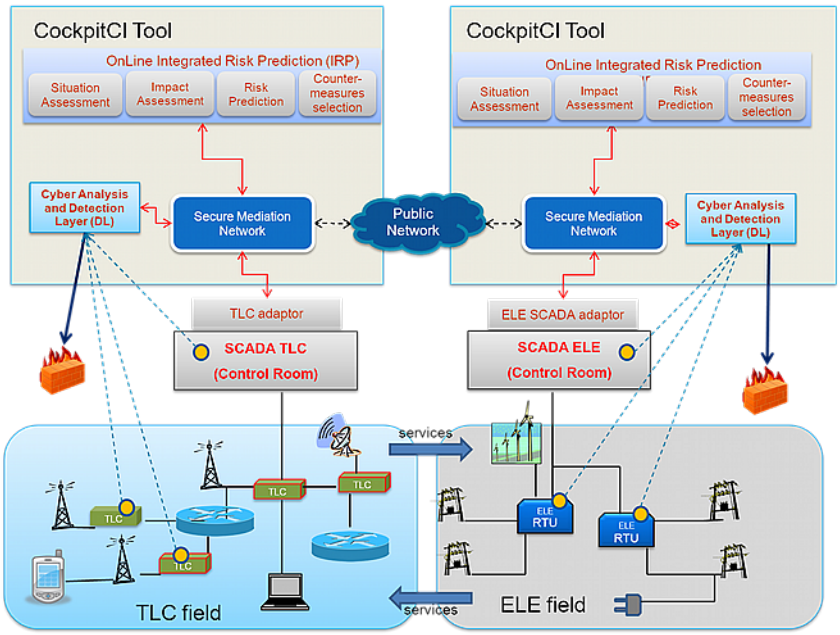

Figure 2.1 – CockpitCI operational concept (from deliverable D3.5).

The operational concept of the CockpitCI, depicted in Figure 2.1 , shows two interdependent CIs, electrical CI (ELE CI) and telecommunication CI (TLC CI), which exchange information between each other, the TLC CI provides telecommunication services to the ELE CI and the ELE CI provides power to the TLC CI. Each one of the CI has a SCADA control room and a CockpitCI tool. The CockpitCI tools collect data from the field and SCADA, via cyber sensing probes and the SCADA adaptor. In order to provide CI operators a better situational awareness over the system of systems in the presence of cyber attacks and, therefore, increase the CI level of service, the two infrastructures share information via the Secure Mediation Network (SMN), which provides secure information exchange via public network. The figure also shows other two components: the Cyber Analysis and Detection Layer (DL) which performs cyber sensing and provides cyber detection capability and the Integrated Risk Prediction Tool (IRP)

which provides situation awareness and assesses risk. The Secure Mediation Network is the only mean by which all CockpitCI system internal components can communicate with their remote corresponding modules.

2.2 Cyber Analysis and Detection Layer

The CockpitCI project is broken down into several work-packages to improve manageability. The University of Coimbra LCT team leads the group responsible for the work-package whose main goal is to conceive and develop the cyber analysis and detection layer (WP3000). As pointed before, this layer provides the detection and analysis capabilities to the CockpitCI.

This layer should integrate several different detection strategies, distributed along different levels. The analysis and detection infrastructure, under development, is designed taking into account local detection mechanisms that are able to function autonomously on each component of the industrial control network, and provide coordinated detection mechanisms, for multidimensional distributed intrusion detection.

The applied detection techniques integrate more classical approaches, such as signature-based IDS tools and classic anomaly-based detection and event correlation, with more advanced solutions, such as machine learning based approaches, including innovative data mining and pattern recognition approaches towards event correlation. These techniques take into account the specific nature of industrial control networks. The idea is to use aggressive usage of topology and system-specific detection mechanisms based on the fact that the role and behavior of each system components are expected to be more consistent over time than other types of networks.

It encompasses detection agents, including adapters for existing intrusion detection systems, as well as specialized network probes and honeypots. That when added to the network are able to capture behavior or traffic patterns. Moreover, it includes a Dynamic Perimeter Intrusion Detection System performing many of the tasks traditionally associated with a Distributed Intrusion Detection System.

The work-package is divided into smaller tasks, with more defined goals, each one with different participant teams.

The University of Coimbra LCT team is involved in several of this tasks with different responsibilities in one each of them. The most relevant are:

- Design of detection agents and field adaptors
- Definition of Real-time intrusion detection strategies
- Design of the Dynamic PIDS
- Implementation and trials

The design, implementation and trial of the Dynamic PIDS can be considered one of the main tasks as it integrates the work developed in the other tasks. The work developed during this thesis is integrated into the work developed by the team in the design, implementation and trial of the Dynamic PIDS as well as collaboration in the other tasks.

2.3 Dynamic Perimeter Intrusion Detection System

In this Section is given an overview of the Dynamic PIDS. The details presented in this section were one of the starting points for the work developed in this thesis.

The architecture of the PIDS, as defined in deliverable D3.1.2 [1] and represented in Figure 2.2 in a simplified way, aggregates several probing and monitoring sensors to provide the surveillance capabilities for the security platform.

The deployment of these sensors divide the SCADA network in three different network security zones:

IT Network while this network is not part of the SCADA network, there are some SCADA components that can be hosted in this zone like Human-Machine Interaction (HMI) consoles. Moreover, historical evidence has shown that several successful attacks reach SCADA components through this level of networking infrastructure.

Operations Network this network hosts the main SCADA components, such as Master Stations, Database Management Systems (DBMS) or HMI consoles.

Field Network in this network are hosted the field devices, such as Remote Terminal Units (RTUs) and process sensors.

As stated in deliverable D3.1.2 [1], the network separation has two purposes. First, to segment different infrastructure contexts for which different detection and correlation strategies might apply. Second, to provide well-defined security perimeters between each zone were mediation mechanisms may inspect and control information flows between each zone.

The main type of security detection sensors considered in the architecture of the PIDS are:

Host Intrusion Detection System (HIDS) are security detection sensors located in individual hosts or devices of the network that gather audit data through analysis of system logs. Additionally, they provide file integrity checking, system configuration checking and other methods to detect abnormal activity that could be an indication of an attack.

Network Intrusion Detection System (NIDS) are security detection sensors placed in strategic points of the network that detect intrusions by analyzing network packets for signs of malicious activity. A signature (pattern) is used to match specific events, such as an attack attempt, to traffic on the network. If the traffic seen on the network matches a defined signature, an event alert is generated.

Honeypot is a decoy or trap that is set up to attract and detect/observe attacks. It simulates a component of the system that appears to be part of the network, usually without providing real functionality, it acts like as a dummy target. Its purpose is to lure and track intruders as they advance to interact with this system; this interaction can be an indication of an attack as the normal functioning of the system does not interact with the honeypot.

Shadow RTU is responsible for monitoring the events handled by the electronic device that interfaces objects in the physical world to SCADA system, the Remote Terminal Units (RTUs).

While some security detection sensors are tools based in existing open source software others are based on research and design by other members of the LCT group integrated with the CockpitCI project and also by other external partner teams members of the project. The design and implementation of the Shadow RTU are the subject of the thesis to be presented this year by another member of the LCT CockpitCI team.

Although not a security sensor the One Class Support Vector Machine (OCSVM) module will provide information to the correlators. This module uses supervised learning models with associated learning algorithms that analyze data and recognize patterns for intrusion detection, therefore, allowing the PIDS to detect unknown attacks. The partner team from the University of Surrey is responsible for developing this module.

The Security Management Platform (SMP) is responsible for managing all the involved components of PIDS. It includes the mechanisms, among others, for monitor and manage detection agents, correlators, OCSVM and feed other CockpitCI systems by retrieving relevant information from the correlation platform.

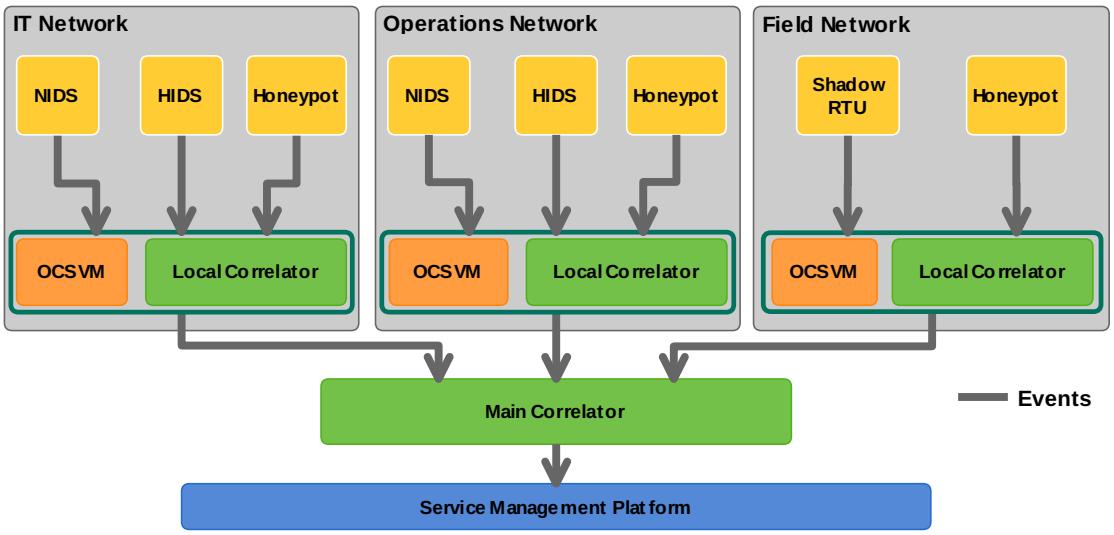

Figure 2.2 – Architecture overview of the Perimeter Intrusion Detection System (adapted from deliverable D3.1).

Chapter 3

Event Correlation: An overview

This chapter gives an overview of event correlation and describes the different techniques used for event correlation. Furthermore, it presents a survey of several existing open source correlation software including a feature comparison. The purpose is to select the best tool to be used in the correlation platform.

3.1 Event Correlation

Intrusion detection systems, like HIDS and NIDS, generate a great amount of alerts that may not be a result of a real alert, called *false positives* [2]. Additionally, the intrusion alert messages they produce may lack information due to the fact these security sensors have very specific domains of operation. For example, a NIDS only sees network-based information while a HIDS does not have much information about the network. If a HIDS reports an alert message about an unauthorized file change, it does not have the information about the IP of the machine connected to the host at the time of that change. Similar examples can be given for the Honeypot and Shadow RTU.

The information provided by the IDS also has limited contextual information, for example, there is limited or non-existent knowledge of the global network topology and other information about the host they are protecting. Hence, does not allow the discovery of a distributed attack to several nodes of the infrastructure.

In order to reduce the limitations of the isolated intrusion detection systems, security correlation systems are used.

Some authors call the process of this correlation systems, event correlation 3; 4; 5, others call it alert correlation 6; 7; 2 and others make a explicit distinction between event and alert correlation 8. In this thesis both terms, event correlation and alert correlation, will be used interchangeably to define the operation that finds the causal relationships between events or alerts.

These systems collect the events from the several security sensors and, as such,

can provide a high level overview of all the infrastructure, combining the information from all this systems.

Event correlation can produce more succinct overview of the security activity of the network. This is achieved by suppressing events that do not provide useful information, aggregating events that refer to the same incident or filtering duplicate events.

3.2 Event Correlation Operations

Event correlation encompasses several correlation operations. These operations differ in the operations executed over the events. Several operations can be combined to provide more complex event correlation patterns. In the next sections are presented the main correlations operations, according to [2; 9; 10]:

3.2.1 Compression

This operation consists in grouping several similar events into one event. In this operation, the events that where grouped are discarded and replaced by a new event. Thus, reducing the total number of events.

3.2.2 Aggregation

This operation collects multiple events and generates a new. The new event as new meaning than the ones it aggregates. In this operation, the aggregated events are not discarded, like in the compression operation. The new event contains references for the events it aggregates.

It can be used to combine events that represent the independent detection of the same attack occurrence by different intrusion detection systems.

3.2.3 Thresholding

Thresholding consists generating a new event if the number of occurrences of a given type reached a certain threshold. In this operation, if the threshold is not reached, the events are discarded.

3.2.4 Filtering

This operation consists in suppressing certain events based on the attributes of the event being discarded. This is considered a stateless operation as no other conditions are taken in consideration, beside each event properties. The filtering can also be used as rate-limiting, this allows to forward events at no more than a given rate, filtering all the others.

3.2.5 Selective Suppression

In this operation, the events are discarded according to the state of the correlator. The event is only suppressed if it meets other criteria, beyond the event properties, like a variable value, the temporal relationship with other events or the presence of other events.

3.2.6 Prioritization

This operation can be used to can be used to change certain event properties, as its priority. This allows to identify the information that is important and the one that is irrelevant. Prioritization can depend on, the state of the correlator, other events or even an external source of information.

3.2.7 Enrichment

This operation consists in augmenting the information of the event. This adds additional information to an event, that is either extracted from other related events or the state of the correlator.

3.2.8 Time-linking

This operation correlates the events based on time and order they arrive. An Example of time-linking is described bellow:

- *Event A* happened within 5 minutes after *Event B* happened;

3.3 Event Correlation Techniques

The main techniques used in event correlation are described in this section.

3.3.1 Rule-based Event Correlation

This type of correlation uses a set of predefined rules to evaluate incoming events until a conclusion is reached. The correlation depends only on the capabilities and comprehensiveness of the rule set. The rules usually have the form *if* condition *then* action. The rule is trigger when an input event together with the state of the system match a condition. The action can also be an input to other rules.

The rules in a system that use this technique are or more or less human readable so that their effect is supposed to be intuitive.

Rule-based systems are only feasible for problems for which any and all knowledge in the problem area can be written in the form of if-then rules and for which this problem area is not large

The rule sets can become very large which may lead to unintended rule interactions and can make the system difficult to maintain the system and can suffer a

performance hit. Additionally, the system is going to fail if an unknown situation occurs which has not been covered by the rules so far.

3.3.2 Codebook-based Event Correlation

This technique, explained in [11], is similar to the rule-based techniques but rather than treating events separately, they are grouped into an alarm vector which represents all of the events. This alarm vector is then matched against problem signatures in a so-called codebook, which represents an optimal set of alarms.

It comprises two stages: an initial stage of pre-processing designed codebook selection, where a subset of alarms is chosen and the second stage, decoding phase, where the alarm vector is analyzed in order to find problems and causes.

This approach as an advantage in comparison with the rule based correlation as it can deal with unknown combination of events.

3.3.3 Case-based Event Correlation

Case-based correlation [12], unlike the techniques presented before, does not need prior knowledge about the infrastructures. Systems that use this technique try to solve a given problem by searching for the most similar case from a base library and retrieve the solution. A case consists of a problem, its solution and, usually, annotations about how the solution was derived. The principle consists in solving incremental problems with a learning component. After applying the new solution to the problem, the result will be verified, and if it is successful, the new case will be stored in the library. Otherwise, a better solution must be proposed, which after validation will be added to the library. As a result, this technique allows systems to learn from experience and can adapt to unknown problems.

3.3.4 Statistical Event Correlation

In [13], a statistical correlation technique that uses a Bayesian network to model the causality relationship between alerts. In this model, alerts are represented by nodes and their causality represented by edges. Based on this model, the goal is to determine which alert types may cause of type A and how the conditional probability of A is related to its parents, the causes. The algorithm proposed is based on mutual information between alerts to find the structure of the Bayesian network, the causal relationship between alerts. When the structure of the Bayesian network is known, is is simple to obtain the Conditional Probability Tables using the statistical relationship between alerts divided along time slots.

3.3.5 Model-based Correlation

The Model-based approach [14] is more a paradigm rather than a detailed technique to correlate events. It refers to the use of a model of the physical world representing the structure and behavior of the system under observation, as an inference

method. Each component of the infrastructure is modeled regarding its attributes, behavior and relation with other models. The behavior of the whole infrastructure is a result of the interaction of the component models. The event correlation is a result of the collaborations of the models comprising the whole infrastructure.

3.4 Existing Open Source Event Correlation Software

In this section, is presented an overview of the features of the available correlation tools, in order to get an early indication of their capabilities and suitability to be use in the correlation platform.

Commercial event correlation software was not analyzed as the purpose of this survey was only to evaluate tools that could eventually be used in the CockpitCI Project and the use of open source software was one of the requirements for the software to use in the PIDS.

3.4.1 General Purpose Correlation Software

This section describes the open source correlation software with a generic purpose. The tools presented here can be used in different scenarios, they are not specific to the security domain. Although some of them were designed with a specific goal, they are generic enough to process and correlate events from different domains.

3.4.1.1 Esper

Esper is an open source event series analysis and event correlation engine. Event series analysis is related to the processing of historical events while event correlation is intended for current-arriving events. *Esper* is not an application by itself but an embeddable component that can be integrated, by using an API, in Java and *JavaEE* applications. There is also a version, called *Nesper*, written in C#, designed to be integrated with .NET applications. *Nesper* contains similar features as Esper. While Esper is an open source Complex Event Processing (CEP) engine, there are two commercial and more complete versions: *Esper* Enterprise Edition and *EsperHA*. Compared to the open source version the commercial versions provide additional features, these include among others: High Availability, GUI for design and management of EPL statements, real-time interactive graphs and charts to display event stream data, REST Web Services for access to CEP and data.

To express different operations correlation operations, *Esper* uses the Event Processing Language (EPL), a declarative language that has similarities with the Structured Query Language (SQL). Like SQL, this language provides `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY` clauses. Streams replace tables as the source of data and events replace rows as the basic unit of data. Since events are composed of data, the SQL concepts of correlation through joins, filtering and aggregation through grouping can be effectively be used.

The following sample EPL, taken from the *Esper* tutorial [15], shows a query that that returns the average price per symbol for the last 100 stock ticks:

```
select symbol, avg(price) as averagePrice
from StockTickEvent.win:length(100)
group by symbol
```

In addition to the event stream processing provided by the EPL language, *Esper* includes pattern matching semantics to define more complex patterns like temporal causality. The pattern matching language is based on a state machine technique. Both the EPL and patterns can be combined together to provide more complex temporal logic.

Below is an example of an *Esper* pattern, from [15], where a property of a following event must match a property from the first event:

```
every a=EventX -> every b=EventY(objectID=a.objectID)
```

Figure 3.1 shows an example of how Event Stream Processing and correlation can be combined among different input event streams.

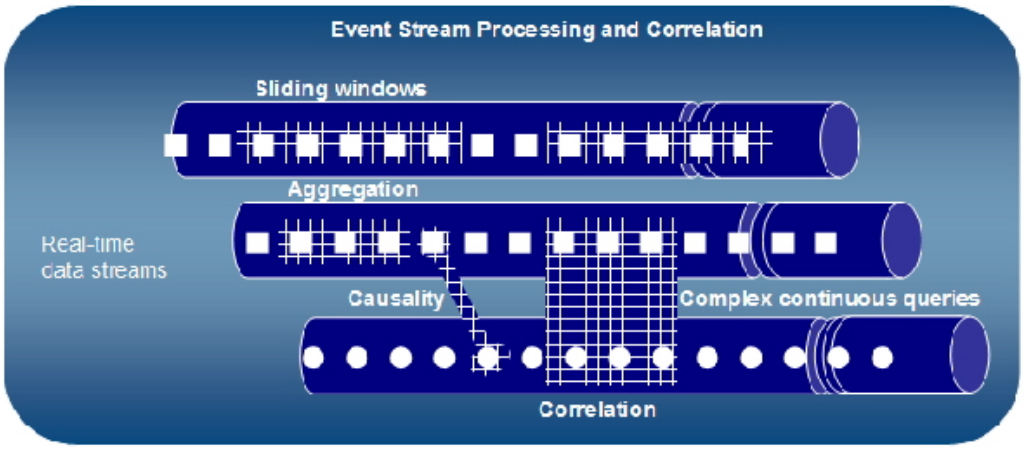

Figure 3.1 – Esper Event Stream Processing and correlation (image from [16]).

As stated in the *Esper*'s FAQ [17] the engine works like a database turned upside-down. Unlike the traditional databases that store data and run queries against this data, *Esper* stores the queries and runs the incoming data through the stored queries. When an event is received, the response from Esper is in real-time if this event matches the conditions defined in any of the queries.

Esper provides a historical data access layer to connect to traditional databases, this enables joining event series with data stored in SQL databases. The idea is to enrich the output with historical data, or use the historical data to further filter stream.

As it is an engine library, an application needs to be built to provide a way that allows *Esper* to receive events. Events accepted by *Esper* can be represented by Plain Old Java Objects (POJO), XML, Arrays of Java Objects or Java Maps.

This correlator engine has a very active community, the discussions are mainly made on the mailing lists [18; 19].

3.4.1.2 NodeBrain

NodeBrain [20] is, at its core, a rule engine with a modular architecture. It includes several modules, called nodes in NodeBrain terminology, that can be used to extend the features of the rule engine. NodeBrain is designed for construction of state event monitoring and event correlation applications by combining the core engine with the some of the modules, according the needs.

A declarative rule-based language is used to express the rules that can be interpreted by the rule engine.

NodeBrain receives can receive input events from regular files, like text log files, syslog or named pipes.

This tool has different operation modes; it can run in interactive, batch, servant or service mode. The service mode allows *NodeBrain* to be executed as a persistent daemon. In batch mode *NodeBrain* is executed like a shell script. The interactive mode allows to access the interpreter directly, somewhat like the Unix shell. Hence, is particular interesting to test new rules and to communicate with nodes in service mode. The servant mode is a temporary child process spawned to handle a system shell command or another function.

Below is presented a description of some of the most useful modules included with *NodeBrain* [21]:

Audit for log file monitoring. It monitors lines of text written to system and application logs.

Baseline module provides statistical anomaly detection. Allows to maintain a simple statistical profile for a set of values to measure. A Baseline node monitors the current value of each measure and alerts when a measure is considered an anomaly relative to the statistical profile.

Cache module provides detection of event repetition, variation or sequence, this enables *NodeBrain* to keep certain values in memory resident table for short-term storage of events. Thus, allowing to perform event correlation by comparing the parameters of current event with parameters of previous events stored in the cache.

Message module that allows broadcasting messages to multiple *NodeBrain* peers, it uses the NodeBrain Message API.

Peer module provides authenticated and encrypted peer-to-peer communication. It can be used for the communication of NodeBrain running in the same or different hosts. The store and forward messaging pattern can be used in the transmission of the messages. This pattern provides reliability to the transmission. If destination system is unavailable, the message is stored locally and forwarded to the remote destination once it becomes available.

Servant is a shell command executed as a child process to *NodeBrain*. *NodeBrain* can communicate with the child process via *stdin* and *stdout*. The idea is to allow

programs written in different languages to communicate easily with *NodeBrain* rules, particularly convenient for scripting languages.

Snmptrap is used to monitor Simple Network Management Protocol (SNMP) traps, it should be noted that however its does not use MIBs.

String module can be used to do simple string manipulation to consume text representations of events.

Syslog enables listening to UDP port for syslog messages, therefore, allowing remote syslog monitoring.

Translator module recognizes elements of foreign text and converts them into NodeBrain commands; it uses regular expressions to recognize and extract elements from input events.

Webster provides a simple web server interface to *NodeBrain*. Allowing to manage (add, edit, delete) *NodeBrain* rules and other simple operations via a TLS encrypted web interface.

A NodeBrain rule associates an action with a condition, the syntax is as follows:

```
define term type(condition) assertion: command
```

The *term* is the name used to reference the rule, *type* is the type of the rule, *condition* is the condition to monitor, the optional *assertion* clause is a set of assignments (for example *counterA=10*). *Command* is the action to be executed when the rule is triggered, the action can be a script or application. There are three types of rules: ON, WHEN, and IF. An ON rule will fire any time the condition transitions to a True value from a non-True value. A WHEN rule is just like an ON rule, except it only fires once. After a WHEN rule fires, it is removed from the interpreter's memory. There is also the IF rule that will always fires when True, but only for a specific NodeBrain command.

From the previously described modules the Cache module is the one that provides features for event correlation where the goal is to detect repetition, it allows to detect patterns of events that, can be described as follows, adapted from the application tutorial [22]:

- *EventA* happened N times within time period P or interval I;
- *EventA* was associated with *EventB* in N events within time period P or interval I;
- *EventB* was associated with N different values of *EventB* within time period P or interval I;
- *EventA* happened within interval I after *EventB* happened;

NodeBrain documentation [23] is very complete and detailed, including many tutorials and examples.

The user community of NodeBrain is almost non-existent and the user mailing list [23] has almost no activity.

3.4.1.3 SEC

The Simple Event Correlator (SEC) [24] is an open source real-time event correlation tool. This tool uses rule-based correlation for event processing. *SEC* is written in Perl, this allows this software to be platform independent. The goal of its author was to create a lightweight and easily customizable tool that could be used for a variety of event correlations tasks [25].

This event correlator can receive events from regular files, named pipes and standard input. The events are recognized from the lines of text, received from the file input stream, by using regular expression language or Perl subroutines. Perl subroutines are functions written in Perl.

The configuration of *SEC* rules is stored in text files. Each of the files can have more than one rule. The rule sets from different files are applied logically in parallel [25]. A rule in *SEC* is normally specified by an event matching condition, a list of actions and optional contexts. The event matching condition is the pattern specified in the rule that is looked for in the input to check if it matches. A context is a logical entity that can be created or deleted from a rule. The lifetime of a context can, optionally, be defined at the context creation [25]. The presence or absence of a context can determine if a rule is applicable or not. This entity can also be used as an event store. If events are associated with a context as they occur, they can later be processed with an external application or written to a file [25].

SEC can be configured with a set of predefined configurable rules. According to Rouillard [26], these rules can be divided into two groups, basic and complex rules. For Rouillard [26], basic rules types are rules that perform actions and do not start an active correlation operation that persists in time. As described in *SEC* man page [27] these basic types are:

Single: immediately executes an action list when an event has matched the rule. An event matches the rule if the pattern matches the event and the context expression (if given) evaluates True.

Suppress: takes no action when an event has matched the rule, and keeps matching events from being processed by later rules in the configuration file.

Calendar: executes an action at specific time and supports repetition, this rule reacts only to the system clock.

The complex rules are described as follows in *SEC* man page [27]:

SingleWithScript: when an input event is matched SEC forks a process to execute an external program. The action to be executed depends on the exit status of the forked process.

SingleWithSuppress: when an input event is matched the action list is executed, but the following matching events will be ignored for the next t seconds. This rule is used to filter repeated instances of the same event for a certain time.

Pair: when an input event is matched the first action is executed immediately. The following matching events are ignored during the next t seconds until some other input event arrives. When the second events arrives another action list is executed.

PairWithWindow: when an input event is matched wait t seconds for another input event to arrive. If the event arrives within the time window t, execute an action list. If the event does not arrive within the defined time window, execute another action list.

SingleWithThreshold: count the number of input events that are matched during the window time of t seconds and execute an action list. If a threshold n is given only execute the action list if the threshold is exceeded. The window is sliding.

SingleWith2Thresholds: count the number of input events that are matched during the window time of t seconds and execute an action list. If a threshold n is given only execute the action list if the threshold is exceeded. The counting of the threshold continues after the execution of the first action list, when no more than n events have been observed during the last t seconds, execute a second action list. Both event correlation windows are sliding.
Below is an an example of *SingleWithThreshold* rule, taken from *SEC* tutorial [28].

```
type=SingleWithThreshold
ptype=RegExp
pattern=foo
desc=$0
action=write - foo matched three times in 10 seconds!
window=10
thresh=3
```

The *action* in the above rule is executed when the pattern *foo* is matched *thresh* number of times in the input data events during the *window* time of 10 seconds.

EventGroup: this rule runs event correlation operations for counting repeated instances of N different events during T seconds and taking an action if the threshold conditions, defined for each one of the N events, are satisfied. The event correlation window is sliding.

Combining one or more of the above rules with appropriate actions and contexts allows to define more complex event correlation schemes.

This tool has large user community and a very active mailing-list [29].

3.4.1.4 Drools

Drools [30] is an open source Business Rule Management System (BRMS) platform. The module responsible for event processing (or CEP) and temporal reasoning is the Drools Fusion [31] module. *Drools Fusion* can be used as an independent module from the rest of the platform.

Like *Esper*, *Drools Fusion* is not an application, but a rule engine software library written in Java. It provides an API that can be used to include this library in applications developed in Java.

This Rule-Based engine is classified by [32] as Production Rule System. These systems use an inference engine that matches facts and data, against Production Rules or just Rules, to infer conclusions that result in actions. *Drools* uses an Object Oriented system optimized version of the *Rete* algorithm [33], to implement the forward chaining method of its Inference Engine. The *Rete* algorithm is designed to sacrifice memory for increased speed [34]. Forward chaining is a reasoning method that starts with the available data (known knowledge) and progresses to a goal state. While progressing from a state to state, all inference rules are fired making all knowledge available within the current state.

Drools Fusion provides two event processing modes, as described below:

Cloud mode in this mode there is no notion of time, neither there is a requirement for event ordering, the engine sees the events as an unordered cloud which tries to match rules. In this mode does not remove events that any longer match a rule automatically, they need to be removed explicitly.

Stream mode is the needed when an application has to process streams of events. As a result, it requires that the events in each stream to be time-ordered, this means that events that happened first must be inserted first into the engine.
If there more than one input stream there is a session clock that forces the synchronization between streams.
As in this mode there is the notion of time it allows rules with sliding window support and also automatic event lifecycle management, where events that no longer match a rule are removed from the engine.

Drools Fusion uses a specific rule language with the following structure:

```
rule "name"
        attributes
  when
        LHS
  then
        RHS
end
```

The optional attributes are hints of how the rule should behave. Left-hand-side LHS is the conditional part of the rule, is a set of productions that contains the unordered sequence of patterns (conditions). The right-hand-side RHS is a block that allows dialect specific code to be executed; it contains the actions. The actions dialect can either be Java or MVEL [35]; MVEL is an expression language for Java-based applications. The rules are usually read from text files containing one or more rules.

Below is presented an example of rule involving a sliding window, in MVEL dialect, taken from the Drools documentation [36]:

```
rule "Sound the alarm in case temperature rises above threshold"
  when
    TemperatureThreshold($max : max)
        Number(doubleValue > $max) from accumulate(
            SensorReading($temp : temperature) over window:time(10m),
            average($temp))
  then
        // sound the alarm
end
```

The rule above represents the situation where an alarm must sound in case the average temperature over the last ten minutes, read from the sensor, is above the threshold value.

Drools has a very active community; the discussion is mainly through the user mailing list [37].

3.4.2 Security Specific Correlation Software

The tools presented in this section are specifically designed to process and correlate security events. Some of them do much more than just event correlation. They are complete Security Information and Event Management (SIEM) solutions. A SIEM combines Security Information Management (SIM), and Security Event Management (SEM) functions into one security management system.

3.4.2.1 OSSIM

OSSIM [38] is an open source version of a commercial product, the AlienVault Security Information and Event Management (SIEM) [39]. It is not considered a product, but a solution that integrates many open source security tools in a single Linux distribution.

A SIEM provides more functionality besides the event correlation and data aggregation, usually it provides functionalities such as a dashboard for event visualization, long-term storage of historical events for forensic investigations and data retention policies.

The integrated tools in *OSSIM* include, among others, the following:

- *Arpwatch*, used for MAC address anomaly detection;
- *P0f*, used for Operative System detection by passive collection of configuration attributes from a remote device during standard layer 4 network communications;
- *PADS* (Passive Asset Detection System), used for service anomaly detection;
- Nessus, used for vulnerability assessment;
- *Snort*, used as an Intrusion detection system (IDS), and can also used for cross correlation with *Nessus*;
- *Tcptrack*, used for session data information which can grant useful information for attack correlation;

- *Ntop*, used for network usage monitoring;
- *Nagios*, used to monitor host and service availability information based on a host asset database;
- *Osiris*, a Host-based intrusion detection system (HIDS);
- *Snare*, a log collector for windows systems;
- *OSSEC*, a Host-based intrusion detection system (HIDS);

This distribution has plug-ins for receiving input from many Linux and Windows sensors. All the tools are linked together in a graphical console that gives the user a single, integrated overview of security-related aspects of the system. The solution enables post-processing of events allowing prioritization and risk assessment.

The open source version as the following limitations comparing to the commercial version, mainly:

- reduced scalability on open source version, no multi-level deployment;
- it is for small, simple, environments that do not require a complex deployment architecture;
- does not support the logger module that provides forensic storage;
- the open source version has more limited speed than the commercial version;
- the web interface is only for local configuration, cannot manage remote components;

The correlation engine of *OSSIM* employs two different correlation methods:

- Correlation using sequences of events: focused on known and detectable attacks by using rules implemented by a state machine. These method is configured through XML files
- Correlation using heuristic algorithms: uses algorithms that attempt to detect risky situations using heuristic analysis. In an effort to compensate for the shortcomings of other methods, these algorithms are intended to detect unknown attacks for which no rules are available.

3.4.2.2 Prelude

Prelude [40] like *OSSIM* provides more functionality than event correlation as it is a SIEM application. This application if focused goal is to collect, normalize, sort, aggregate, correlate and report all security-related events.

Prelude has a distributed modular architecture [41], the main module is the *Prelude-Manager* a centralized server that accepts secure connections from distributed sensors or other *Prelude-Managers*, it acts as an event concentrator that also saves the

received events to a persistent storage. Using plug-ins this module also provides event filtering, event suppression, event thresholding and normalization.

The intrusion detection is provided by sensors that report security events to the Prelude-Manager.

All event communication between the different Prelude modules is done in a single format the Intrusion Detection Message Exchange Format (IDMEF), this format is detailed in Section 5.3.1.

The *Prelude-Correlator* [42] module allows multistream event correlation. It is a Python based rule engine, the rules are classes written in Python. The module connects and fetches the events from the Prelude-Manager server and correlates the incoming events based on the rule set. When a rule generates an alert, an IDMEF event is sent to the server. The module already is distributed with a default correlation rule set.

The Prelude-LML module provides log analysis capabilities to *Prelude*. It monitors log files, logs from *Syslog* daemon or other types of single-line event logs. Uses regular expressions to analyze and monitor the logs. The tool is distributed with an extensive list of default regular expression rules for security and logging applications.

A web-based graphical user interface is provided by the *Prewikka* module[43], it provides a graphical interface for permission management, event alert listing, sensor monitoring and filter creation.

Prelude OSS, the open source edition of *Prelude* has some limitation when compared with the commercial Pro edition. The open source edition *Prelude OSS* is aimed for evaluation, research and testing purposes on small environments. The performance of this version is much lower than the *Prelude Pro* edition.

3.4.2.3 Sagan

Sagan [44]is a multi-threaded, real-time Security Event Management and Analyzer application. *Sagan* uses rules similar to *Snort* [45] rules to detect malicious traffic. Snort is a widely used Network Intrusion Detection System (NIDS) and its rule format its a well know in the security field.

Sagan has the capability to monitor events from different sources by using FIFO and *stdin* as input, this allows to use this tool with any application that is able to write to a FIFO or standard output.

As *Sagan* uses *Snort* syntax rules it is easy to write new ones for those used to *Snort*. The application already includes an extensive list of predefined attack rule signatures.

Although this application can run without *Snort,* its usage is tightly coupled with *Snort*, *Sagan* only does correlation with *Snort* events. To correlate events from *Sagan* with those of *Snort,* the application must use *the Snort* database to save the events. These events are kept as a different sensor from those from *Snort*. The collected by *Sagan* from the different sources are then correlated with *Snort* ones.

The correlation is not configurable and is done by the application automatically by using the following methods:

- Time stamp
- Destination TCP/IP address
- Source TCP/IP address
- Destination TCP/IP port
- Source TCP/IP port
- Protocol used
- Classification

As this tool write the events to the a Snort database, it is possible to use a Snort console to also visualize the Sagan events, this allows to monitor packet level threats and log level events from a unified console.

3.4.2.4 ACARM-ng

Alert Correlation, Assessment and Reaction Module-next generation (ACARM-ng) [46] is a system for correlation of security alerts. It aims to collect and correlate information coming from IDS components located in the network infrastructure.

Designed to work in conjunction with *Prelude-Manager*, albeit it can work without it. The *Prelude* module act has hub in a multi-level configuration and data gathering point from multiple sensors.

This application accepts inputs from *Prelude-Manager* or IDMEF XML files; other input modules can be written.

The main application is easy to install without requiring many dependencies. However, some of the optional modules required some patching of source code. The *Prelude-Manager* dependency is the one that requires most of the work.

ACARM-ng has a modular architecture made of plug-ins. It includes a pre-processor, this tool allows to remove certain alerts just after receiving them, this is useful when there are known facilities that generates useless alerts that are not intended to be correlated. The pre-processor is configurable by using rules in the form of logical expressions.

Other included plug-in is the filter plug-in, filters are the key components for alert processing and have some configurable parameters. Although most of the filters take part in the process of joining similar alerts, there are some filters as IP blacklist or DNS resolver that can change input alerts by increasing priority or filling DNS names of involved hosts. The following filters are included in the application:

- One to one (joins all the alerts between a given pair of hosts, correlates attacks from one host to another);
- One to many (correlates alerts coming from one source host and with multiple targets. This is a typical scenario of one attacker trying broad network recognition);

- Many to one (correlates attacks performed by *botnets* against a single host in monitored network);
- Many to many (correlates attacks performed by *botnets* against a set of hosts in monitored network);
- IP blacklist (increases priority of alerts generated by attacks coming from one of IPs which are known to be attackers. A list of such IP addresses is downloaded regularly from by this filter);
- Event chain (correlates chains of alerts where one host acts as a source in one alert and as target in another. It can find break-in where captured computer is used as a base for following attacks);
- DNS resolver (performs reverse-DNS mapping on hosts that do not have DNS name set);
- Same name (correlates multiple attacks with the same name);
- User monitor (filter correlating actions of each user)
- Similarity (correlates events similar to each other, above a given threshold; all data is taken into consideration during comparison of two elements)
- New event (changes priority of events, that were not previously seen on the system);

New filters can be written in Python to perform correlation.

The modules responsible for taking predefined actions for input or correlated alerts are called triggers.

ACARM-ng includes triggers to send e-mail notifications, write events to files, invoke external scripts or applications and logging to a console, *Syslog* or files.

3.5 Feature Comparison of the Correlation Software

To provide a better overview of the tools presented in the previous sections Table 3.1 and Table 3.2 shows a summary of the features for all correlators described in the previous sections.

Reviewing the features of the correlators described in Section 3.4 and summarized in tables 3.1 and 3.2 it was found that some of them were not suitable to be used within the correlation platform of the PIDS, as they do not fulfill the requirements for the correlator of the PIDS.

In *OSSIM,* multilevel deployment is not supported by the open source version [47]. As a result, it cannot be used in the two-level correlation architecture of the PIDS, described in section 2.3. Moreover, this tool is distributed as a ISO image designed to be deployed as the main operative system of the host. Thus it is less flexible and harder to integrate with the CockpitCI test bed. For these reasons, *OSSIM* was not considered as an option for the correlation engine to be used.

Correlator	**Language**	**Developed since**	**Last stable release (10/1/2014)**	**License**	**Operative System**	**Real-time**	**Built-in Persis-tence**	**Event communi-cation**
Esper	Java	2004	4.11.0 (January, 2014)	GNU/GPLv2	Cross-platform	yes	no	no
NodeBrain	C	1998	0.8.15 (May, 2013)	GNU/GPLv2	Linux, Windows, MAC OS X, UNIX	yes	no (only to text files)	yes
SEC	Perl	2002	2.7.4 (June, 2013)	GNU/GPLv2	Cross-platform	yes	no (only to text files)	no
Drools Fusion	Java	?	6.0 (December, 2013)	APL	Cross-platform	yes	no	no
Acarm-ng	C++, Python	2011	1.1.1 (May, 2012)	GNU/GPLv2	Linux	yes	yes	yes
OSSIM	several modules written in different languages	2003	4.4 (December, 2013)	GNU/GPL	Linux	yes	yes	yes
Prelude	C, Python, Lua	1998	1.1.1 (September, 2013)	GNU/GPL	Linux, OpenBSD, FreeBSD, NetBSD, Sun/Solaris, MacOSX	yes	yes	yes
Sagan	C	2009	0.3 (April, 2013)	GNU/GPLv2	Linux,OpenBSD	yes	yes (Snort database)	no

Table 3.1 – Open source correlation software features summary.

Correlator	Type of software	Input Event format	Correlation	Management/event visualization GUI	Documentation	User community	Available input sensors
Esper	library	POJO, XML, Java Maps, Java Objects	rule based	no/no	very complete and detailed	very active mailing lists	-
NodeBrain	application	raw text, Syslog	rule based	no/yes (very basic)	very complete and very detailed	no active user community, almost empty mailing lists	Syslog, log files,named pipes
SEC	application	raw text, Syslog	rule based	no/no	complete and detailed	very active mailing lists	log files, named pipes
Drools	library	POJO	rule based	no/no	very complete and detailed	very active mailing lists	-
Acarm-ng	application	XML (IDMEF)	rule-based	no/yes	incomplete and scarce	no active user community, empty mailing lists	uses Prelude-Manager (same sensors than Prelude)
OSSIM	Virtual Machine appliance	OSSIM specific text format	rules, anomaly detection	yes/yes	complete and detailed	very active mailing lists	OSSEC, Snort, Suricata, OpenVAS, osvdb, Nmap, Prads, Nagios, tcpdump, ntop, nfdump, fprobe
Prelude	application	XML (IDMEF)	rule-based, statistical correlation	yes/yes	complete and detailed	no active user community, empty mailing lists	Auditd, Nepenthes, OSSEC, Snort, Suricata, Samhain, Nepenthes, PAM, ufwi-filterd, Sancp
Sagan	application	raw text/syslog	rule-based	no/yes (using a Snort console)	complete and detailed	very active mailing lists	Syslog, Snort

Table 3.2 – Open source correlation software features summary (cont.).

Sagan was also excluded as an option for the correlation engine to be used given that its correlation capabilities are very limited. *Sagan* correlation is not configurable. This tool only correlates events from logs with those of *Snort* based in predefined parameters like: time stamp, destination and source address and port, protocol and classification.

Although *Prelude* satisfied many of the features required for the correlation platform, the open source version *Prelude OSS* is aimed for evaluation and testing purposes in very small environments with performances that are very lower compared to the *Prelude Pro* edition, like stated in Prelude open source web page [40]. Additionally, the development of the open source version advance very slowly, although there was a release in September of 2013 with small updates, the previous one was more than a year before. The mailing lists [48] are almost empty, meaning that the community around the project is almost non existent. Based on the reasons mentioned above *the Prelude* open source version was also not considered as a valid option for inclusion in the correlation platform.

During the analysis, it was found that *ACARM-ng* was also not an option to consider. This tool has no releases since May of 2012. It has very poor documentation and the community is almost non existent, as can be verified by empty forums/mailing lists [49]. *ACARM-ng* sensors are also based in *Prelude-Manager*, thus, have the same limitations than *Prelude* pointed before. For these reasons, this tool was also not considered.

Chapter 4

Communication for Distributed Event Correlation Systems

The distributed correlation platform needs a mechanism to exchange the events between the different components of the platform. In this Chapter are described several technologies for event communication in distributed systems.

4.1 Communication Models for distributed Applications

Two of the most important communication paradigms are the Remote Procedure Call (RPC) and asynchronous messaging using a Message Oriented Middleware (MOM) [50; 51].

In Section 4.1.1 and Section 4.1.2 both paradigms are briefly explained and their suitability to be used for the event communication in the correlation platform is analyzed.

4.1.1 Remote Procedure Call

RPC is procedure or function-oriented interaction model, allowing synchronously to request a remote service execution, where both parties communicate directly. It is suitable to be used in client-server based applications. The systems are tightly coupled by working on functions interfaces or objects.

This paradigm requires simultaneous availability of all subsystems. As the communication is synchronous, the receiving server must be available to accept messages sent. If the server is down, the message cannot be delivered at that time, and it is lost.

Systems built with the RPC model are interdependent, as such RPC provides an inflexible method of integrating multiple systems [51].

4.1.2 Message Oriented Middleware

Message Oriented Middleware [52] is a middleware infrastructure that provides messaging capabilities, allowing communication between disparate software entities. This message-centric approach fits well with the security event transmission, as the unit of information to exchange is the message itself.

In this infrastructure, each client connects to one or more servers, called brokers, which act as an intermediary in the sending and receiving of messages. Thus, the applications, senders and receivers does not know about each others making them be loosely coupled.

Client applications that send the messages are called publishers or producers, while client applications that receive the messages are called consumers. Each of these components, publisher, consumer and broker, can have multiple instances each of them can reside in independent hosts.

One of the strengths of this system is that it allows for efficient communication between applications situated in heterogeneous operating system and networks[53], as such it fits well in a heterogeneous system where the Dynamic PIDS is going to be deployed. The correlation platform needs to receive events from system running in different operative systems, with applications written in several programming languages and developed by different teams of the CockpitCI project. Additionally it should receive events form different networks. Thus, by using a MOM, instead of enabling explicit connections to varied systems and networks, the client applications only need to communicate with the Message Oriented Middleware.

Messaging applications use a client Application Programming Interface (API) to communicate with the MOM. A client application can either act as a sender (producer) that produces messages, or a receiver (consumer) that consumes messages.

The most common MOM implementations use asynchronous message delivery between unconnected applications via a message queue framework, although there are MOM implementations that work without a queue. The queues provide temporary storage when the destination program is busy or not connected.

The MOM provides different messaging models, two of the main ones are the point-to-point and publish-and-subscribe.

In point-to-point (p2p) messages from a producing client are routed to the consumer via a queue. There can be several publishers to the queue, but usually there is only one consumer. Although it is not a requirement, for example, several consumers can be used as load balancing in the consuming side of the system. In this mode, the messages are always delivered and will be stored in the queue until they are consumed by the consumer.

The publish-and-subscribe (pub/sub) is mainly intended to be used as one-to-many and many-to-many broadcast of information. It allows one publisher to send messages to one or several consumers. In this model, the publisher does not need to know about the consumer application. It just sends the message to a destination in the broker. The broker will then send it to the consumer. Publishers can send messages to a specific topic in the broker and only consumers subscribed to that topic are going to

receive the message, but all subscribed consumers receive the message.

A combination of publish-and-subscribe and point-to-point can coexist in a broker, allowing very flexible configurations.

4.1.3 Conclusion

As pointed before, being message centered, the MOM is more targeted to provide security event exchange than the function based RPC model. Additionally, the inflexibility of the RPC, due to its tight coupling and synchronous communication, can be problematic in the heterogeneous systems where the PIDS is going to be deployed. An advantage of the RPC has over MOM is that it can guarantee sequential processing, but in the scope of event communication this is not required.

For geographically dispersed deployments, like the ones that are the target of the PIDS, with strict demands in robustness, reliability, flexibility, and scalability the MOM is the best solution [51; 50].

4.2 Message Oriented Middleware Technologies

In this section are described some open Message Oriented technologies. Not all technologies are listed, but is given an overview of the widest spread technologies, with potential to be used in the correlation platform.

4.2.1 Simple Text Oriented Messaging Protocol

The Simple Text Oriented Messaging Protocol (STOMP) [54] is a lightweight and simple human readable text messaging protocol. It provides an interoperable wire format to allow clients (publishers/consumers) to communicate to any message broker that supports the protocol. This protocol is based on the HTTP protocol. The messages consist in a frame header with properties and a frame body.

This protocol does not deal with topic and queues, the semantics and detailed syntax of the destination tag are not defined in the official specification [55][56], as such, different brokers can interpret the destination in a different way. Hence, interoperability of the protocol can be compromised when using different brokers.

This protocol as several open source implementations available for clients and brokers. The implementations provide libraries in different programming languages.

4.2.2 Message Queue Telemetry Transport

The Message Queue Telemetry Transport (MQTT) [57] is a lightweight broker-based publish/subscribe message-centered wire protocol. It was designed to be used in constrained environments, like embedded systems, mobile devices or sensors. Usually this environments have low bandwidth limited connections, limited processor or limited memory.

It is optimized for the use case where routing of messages is made for simultaneous connected publishers and subscribers [58], as such, it is not suited for the case where the consumer is not connected and the messages need to wait in the queue, although it can be configured this way.

This messaging protocol is agnostic to the payload content, as such, publishers and consumers need to agree on how data is serialized.

There are available several open source implementations of this protocol either for broker or clients in different programming languages.

4.2.3 Java Messaging Service

Java Messaging Service (JMS) [59] is a standard that specifies an API for MOM; it does not specify a wire protocol. This vendor agnostic Java Community Process (JCP) standard defines the interfaces and semantics on how an application can create, send, read and receive a message. A broker is used to route the messages from the publishers to the consumers.

JMS provides a standard for interoperability only within the Java platform. Being a Java API standard and not defining a message format makes the integration with other languages difficult to implement.

4.2.4 Advanced Message Queuing Protocol

The Advanced Message Queuing Protocol (AMQP) [60] provides an open standard application layer protocol for MOM. The AMQP standard was designed with the following main characteristics as goals [61]: security, reliability, interoperability, standard, open.

AMQP is a wire-level protocol, it defines a self-describing encoding scheme of byte sequences to pass over the network. It does not constrain data to be exchanged to a specific format. This type of protocol also provides interoperability among different AMQP compliant software. The specification of the protocol enables conforming client applications to communicate with conforming messaging middleware brokers. Message exchange can be synchronous or asynchronous. It should be noted that different versions of the protocol specification are not interoperable, and applications do not implement all versions of the protocol.

The protocol defines several queuing mechanisms including support for store-and-forward, this allow to the broker to queue the messages when the consumer is not available. Messages will be delivered when the consumer becomes available.

AMQP provides several routing mechanisms. It has the concept of a routing engine, called exchange. Exchanges are entities to where the messages are sent, the messages sent to an exchange are routed to queues by bindings. Bindings are rules that allow the broker to know to which queue the messages is going to be sent. A broker can have one more message queue and exchanges. This allows it to support messaging models beyond the point-to-point and publish-subscriber.

The protocol is vendor-neutral and platform-agnostic. There are several open source implementations for many different programming languages.

4.3 Message Oriented Middleware Comparison

In this section, the MOM protocols presented before are compared and their suitability to be used for the communication of the events is analyzed. In Table4.1 is presented an overview of the key features of each of the technologies.

MQTT and STOMP were designed to be very simple, as a result they are much less flexible than AMQP and JMS. But the flexibility provided by the AMQP exchanges makes it even more flexible than JMS. The flexibility in the routing mechanism can allow to build a more flexible correlation platform.

The fact that JMS only defines an API for Java applications and not messaging protocol sets this technology apart from the others. This limits the interoperability of this technology. Although AMQP, MQTT and STOMP are wire formats the AMQP provides better interoperability than STOMP and MQTT. The limited interoperability of JMS excludes it from being a viable solution for the correlation platform as it does not adapts to heterogeneity of the agents to connect to the correlation platform. The security agents can be written in different programming languages, furthermore, there are applications developed by external teams that are going to connect to the platform. There should be no imposition to use Java.

Not being a standard limits the suitability of STOMP to be used as MOM for security event communication. Additionally its simplicity make its too inflexible to be used in the platform.

According to [58], MQTT is more suited for the case where simple clients connect to a server. It also claims that AMQP supports much more use cases, and that it provides better security and message reliability.

It should be highlighted that many broker implementations support both the MQTT, STOMP and AMQP protocols. In the scope of the correlation platform this can allow to use a different protocol for the limited resources agents, like the ShadowRTU and the Honeypot and have another protocol for the communication between the other components.

	AMQP	**JMS**	**MQTT**	**STOMP**
Type	p2p, Pub/Sub, other	p2p and Pub/Sub	Pub/Sub	p2p and Pub/Sub
Architecture	Brokered	Brokered	Brokered	Brokered
Interoperability	Yes	No	Partial	Partial
License	Open Source	Open Source	Open Source	Open Source
API/Protocol	no/yes	yes/no	no/yes	no/yes
Security	SASL authentication and TLS for data encryption	Vendor Specific usually based on SSL and TLS	Simple username/password authentication, SSL for data encryption	SSL or TLS
Encoding	Binary	Binary	Binary	Plain Text
Transport layer	TCP	Not specified, usually TCP	TCP	TCP
Platform-agnostic	yes	yes	yes	yes
Standard	yes (OASIS)	yes (JCP), but only API	proposed (OASIS)	no

Table 4.1 – MOM features comparison.

Chapter 5

Proposed Architecture

This chapter discusses the formal design of the event correlation platform within the Perimeter Intrusion Detection System (PIDS) of the CockpitCI. The requirements are presented in Section 5.2.

5.1 The Correlation Platform Within the Perimeter Intrusion Detection System

According to the CockpitCI project deliverable D3.1.2 (Requirements and Reference Architecture of the Analysis and Detection Layer [1]), the PIDS must be able to aggregate, filter and analyze information of potential cyber-attacks induced on SCADA systems or telecommunication involved in the operation of Critical Infrastructures, identifying the potential insecurities and vulnerabilities.

The correlation platform is responsible for collecting the events generated by the different security sensors, analyze them in order identify threats and generate alarms. The design, implementation and trial of this platform are the main goals of this thesis.

The correlation platform, as shown in Figure 2.2, comprises two types of event correlators, arranged in a two-level hierarchy. It consists of one local correlator located in each network zone and a main correlator with a global perspective of the global SCADA infrastructure.

5.2 Correlation Platform Requirements

As already mentioned in the section 2.3, the correlation platform proposed here is integrated with the PIDS, therefore, the architecture must fulfill the requirements of this intrusion detection system. Additionally, new requirements specifically related to the event correlation platform were defined. The main requirements are listed below:

- All the components of the correlation platform shall be based on existing open-source software to minimize the costs of development/implementation.
- The global correlator shall be able to send alerts to the Security Management Platform in case of an attack.
- The alerts sent to the Security Management Platform shall be clear, and include information complete as possible and easily understandable by the SCADA operators (identify time, attack, zone, etc).
- The correlation layer shall be able to collect the events generated by the detection layer, analyze them in order to quickly identify threats (according to some predefined rules) and generate alarms.
- The system shall provide a mechanism for event aggregation and event filtering in order to reduce the number of duplicate events, non-critical events and false alarms.
- The local correlators shall be able to send relevant events to the global correlator in case of an attack. The scope of the correlation of each of the local correlators shall be limited to the corresponding network zone.
- The correlation layer shall be a distributed two-level correlation architecture. There shall be a local correlator for each of the network zones of the PIDS. the local correlators shall be capable of processing the events from the agents were they are located. The main correlator shall be capable of correlating events from the different network zones of the PIDS. These events are received from the corresponding local correlators and the OCSVM module.
- It shall be possible to add, edit, delete and view the correlation rules of the correlators.
- All the events sent to the correlator shall be in normalized format.
- The behavior of the correlation shall be deterministic. The behavior shall only depend on the input events, the rules and the internal state of the correlator. The internal state of the correlator, in turn, shall only depend on past events and rules. If external sources of information are used, that are not under control of the correlation engine and can be time dependent, the correlation process cannot be guaranteed to be reproducible.
- The communication protocol shall allow the implementation of lightweight event producer capable of running in a limited resource system (like a *Raspberry Pi*).
- The detection of anomalous security events on the different hosts shall be made by a specialized application, agents (Honeypot, HIDS, NIDS, Shadow RTU).
- All event communication shall be encrypted from source to destination. Only authenticated applications can publish events to the platform.
- The protocol used by the communication layer shall provide interoperability.
- The correlation platform shall be resilient, and the loss of events shall be minimized or even eliminated.
- The correlation application shall be a scalable solution that can have more sources of events added, as needed, without a large impact in global performance.
- The communication and correlation layer should provide fail-over mechanisms.

5.3 Architectural Design

In this section is presented an overview of the distributed security event correlation platform architecture. The platform must be tightly integrated with the architecture of the Dynamic PIDS. As a result, in this thesis is proposed an architecture for several components that define the PIDS. The high level architecture of the PIDS, presented in section 2.3, does not define in detail the several components, it is a global and simplified overview. In this thesis is defined and detailed the architecture of the event communication and event correlation.

As pointed in section 2.3, the correlation platform incorporates two types of correlators arranged in a two-level hierarchy.

This arrangement increases the scalability of the correlation system, as each one of the local correlators is only responsible for processing the events generated by the agents placed in their corresponding network zone, and the main correlator does not process all the events generated by all sensors. The disposition of the correlators in different zones allows to have specialized rule configuration, for each of the zones. Hence, it will limit the total number of correlation rules for each of the correlators, as such it will have a positive impact on the correlator's performance as a large number of rules can decrease its performance.

The difference and specific details of each one of the two types of correlators is described below:

Local correlators report information to the higher level of correlation, also performing event reduction and synthesis (for instance, using duplicate elimination). These correlators act as a data supplier for the main correlator. By performing event reduction and aggregation they send fewer events to the main correlator than those received from all the detections agents. They will behave distinctively according to its network zone, thus allowing to detect specific problems to a particular zone.

Main correlator that is placed above the local correlators, gets a global perspective of the whole SCADA infrastructure, receiving events from each local zone correlator. Due to the broad view of the whole infrastructure, this correlator has an important role in detecting network traversal attacks. This type of attacks happens when an attacker penetrates successive networks layers, like one that starts in the IT network and progresses to Field Network.

The platform is composed by three main elements, each one of them with a different role in the system. A brief description of each of the components role is presented below:

Detection agent components are the security sensors, distributed over different hosts and networks, these components are responsible for monitoring, collecting information and sending security events to the correlation components.

Correlation and analysis components provide a way to extract meaningful information from the events collected by the detection agents by correlating the data provided by the events. It comprises the local and main correlators.

Event communication layer allows reliable and secure communication of events from the detection agent components to the correlation components as well as sending the events to the Security Management Platform (SMP).

5.3.1 Event Format

In a distributed environment like the correlation platform, with heterogeneous security sensors and Intrusion Detection Systems, it is fundamental to specify a common event format for providing interoperability. Additionally, in a platform developed in collaboration with several teams working in a geographical dispersed environment, agreeing on a common language with a well-defined standard is fundamental. Moreover, high level analysis like event correlation also requires events to be processed in a generic format [6].

As such, the data format chosen to represent the information exchanged with the different components of the correlation platform is the Intrusion Detection Message Exchange Format (IDMEF) [62]. This experimental standard defines data formats and exchange procedures to be used by automated intrusion detection and response systems, as well as their management systems. The IDMEF standard is transport independent.

A XML implementation for this standard is defined in the IDMEF RFC draft [62].

The data model used by the format addresses several problems associated with intrusion detection data, as indicated in [62]. As an object-oriented model can provide a flexible way to represent the information from heterogeneous events alerts. It defines classes to represent different intrusion detection environments, like, for example, a NIDS that detects attacks by analyzing network traffic and a HIDS that detect attacks by analyzing logs and files. Can be extended by subclassing or association of new classes to accommodate the difference in sensor capabilities and the way attacks are reported in different operative environments. An overview of the IDMEF data model included in Appendix A and two examples of events attack alerts are included in Appendix B.

Although IDMEF never became a standard its the format used by several intrusion detection software like Prelude [40] and ACARM-ng [46], there are libraries to integrate the format with other tools like the LibIDMEF [63].

As pointed before, using a common data exchange enables interoperability of the correlation platform with other components, as well as making easier to add new elements to the platform, like new security sensors, as the correlation engines will understand the events.

Alternatively there are several proprietary standards. One of them is the Security Device Event Exchange (SDEE), this is an IDS alert format and transport protocol specification defined by a consortium of companies. In addition to no being freely available, the specification is mainly only used by Cisco products.

5.3.2 Detection Agent Components

The detection agents are the security sensors responsible for monitoring, collecting information and sending security events to the correlation components.

The work to be developed regarding the detection agents concerns the design and development of adapters for existing sensor applications that provide detection mechanisms to the PIDS. The adapters ensure the interoperability between heterogeneous agents and the infrastructure of the PIDS. The idea is to develop a component to connect two systems that use distinct information formats. The adapter will make possible to connect systems like *Snort*, *OSSEC* and others with the event communication layer so that these tools can send events in a format (IDMEF) that can be understood and processed by the correlation components.

Most often the different agents report security events, information of potential threats and vulnerabilities to log files. The adapter should capable of capturing this information, normalize the events the from the agent custom format to the IDMEF and publish the events to the communication layer, in order be sent to the correlation components. Additionally, the adapter can provide a simple filtering and aggregation capabilities to reduce duplicate events.

5.3.3 Correlation and Analysis Components

The architecture of each of the two correlators is similar for the local and main correlator. This uniformity allows for easier integration with the communication layer, as the adapters will be the same, moreover, using the same correlator engine allows for expressing the correlation operations using the same language. Thus, simplifying the task of rule management by operators and security experts. Although, both types of correlators have similar architectures, they will have different configurations as well as different correlation rules.

An overview of the correlator internal architecture is depicted in Figure 5.1. The correlator will interface with the communication layer via an input and an output adapter. The input adapter consumes IDMEF events from the communication layer and inserts them into the Esper correlation engine core whereas the output adapter inserts the events generated by the correlation engine into the communication layer to be sent to the upper level correlator or the SMP, according to the type of correlator.

For security auditing purposes, the correlator will log all events and traces of the actions performed to a persistent storage. The events will be logged as they are received in the correlator and the actions executed by the correlator shall also be logged.

The correlation can make use of information taken from external sources. These sources can provide additional information related, among others, to the definition of the network topology, detailed system information. These external sources (knowledge and topology databases) can be queried directly from the correlator core engine.

The management adapter allows to manage and access the event storage database, the knowledge and topology database, the rules, as well as to manage the correlator engine configuration. The management is done from a remote central location, the SMP

New rules should be easily added to the correlation engine dynamically, without the need to restart the application. The design and development of this management

adapter are out of scope of this thesis. The component is in development by another member of the LCT CockpitCI team.

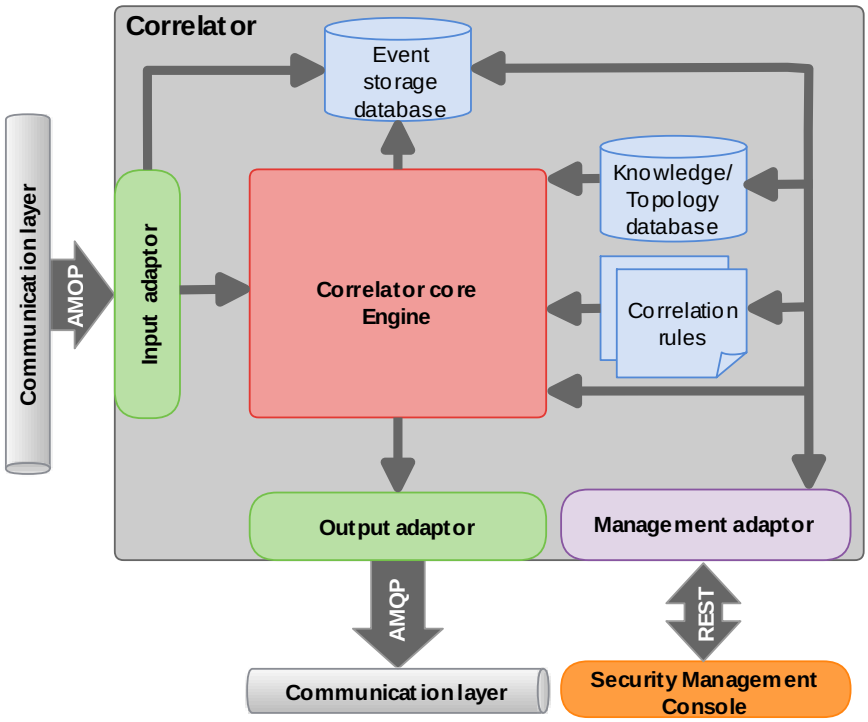

Figure 5.1 – Correlator architecture overview.

5.3.4 Event Communication Layer

The event communication layer is responsible for the transmission of the security events between the different components of the correlation platform in a secure and reliable way.

5.3.4.1 Event Communication Protocol

Based on the comparison of the different MOM technologies, described in section 4.3, the protocol selected to be used by the event communication layer is the AMQP protocol. This protocol fulfills all the requirements for the event communications in the platform.

As pointed before, AMQP is a wire-level protocol, it does not constrain data to be exchanged to a specific format allowing the events in the PIDS to be exchanged in the IDMEF format.

Bellow are described the main feature this protocol can provide to the platform's communication layer :

Security The standard specifies security layers OASIS [64](Transport Layer Security (TLS) and the Simple Authentication and Security Layer (SASL)) that are used to

establish an authenticated and encrypted connection over which regular AMQP traffic can be tunneled.
The security mechanisms provided by the AMQP can ensure that the events are not tampered, that the event is originating from a certain agent and that an attacker cannot know that certain alarms are being generated.

Reliability systems supporting AMQP can guarantee message persistence by using a store-and-forward mechanism. This mechanism is used for persisting events to disk to ensure that they can be recovered if there is a failure in either the messaging system or the consuming client. The events that were sent while the system was unavailable will be redelivered to a system at later time.

Scalability and high availability several AMQP open source broker implementations support clustering [65; 66; 67], this allows a group of brokers to act as a simple broker. The brokers, in a cluster, may run on the same host or different hosts. A cluster can be used to provide high availability and/or scalability/load-balancing. In the high availability configuration, the cluster members replicate state an if one member fail clients can fail-over to another. To provide scalability, the workload is distributed across the multiple brokers that compose the cluster. Although the two goals are different, the configurations can be combined to provide a highly available and scalable system, but providing reliability has a cost because replication is extra work in addition to the normal operation of a broker.

Flexibility AMQP can also provide the publisher-and-subscriber, point-to-point, and other configurable routing models that allow for a more flexible configuration of the platform.

5.3.4.2 Event Communication Architecture

An overview of the architecture of the communication layer is detailed in Figure 5.2.

The events generated by the different detection agents, after being normalized to the IDMEF format, are sent to an event broker through an adapter that connects to the broker. The broker is then responsible to route these events to a queue where the local correlator can consume this events. After processing and correlating the events, each of the local correlators sends the events to another broker from where the main correlator consumes them. The events produced by the main correlator are sent to the broker that routes them to a queue where they can be sent to the Service Management Platform.

The communication system should allow automatic reconnection in case of loss of connection.

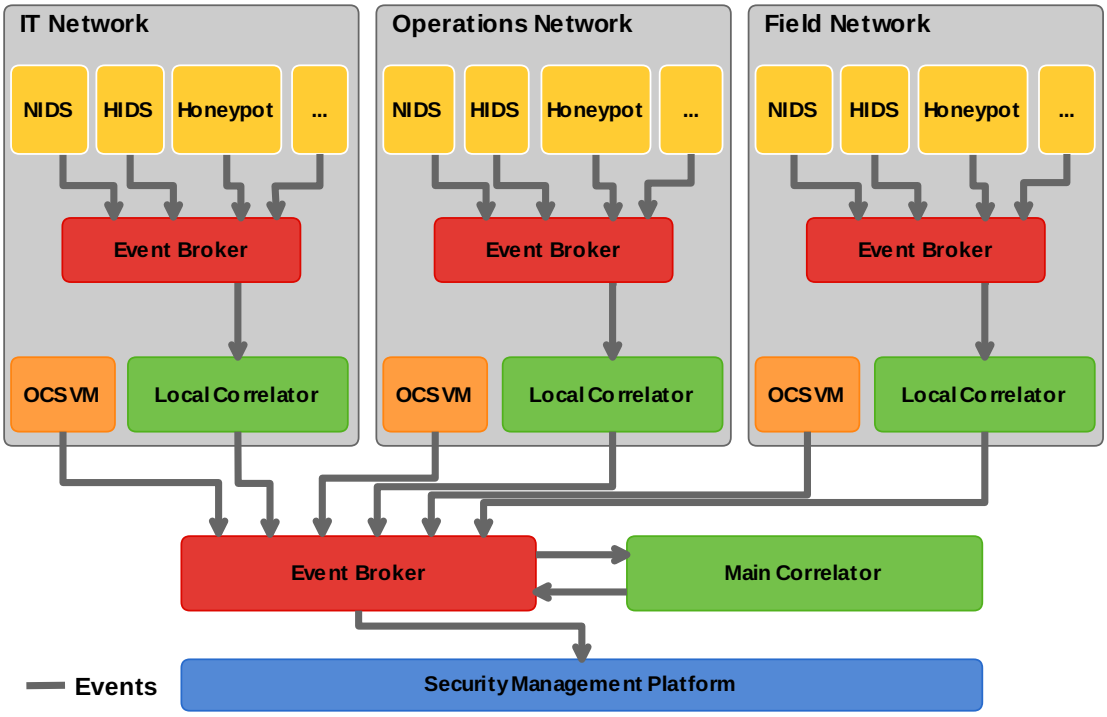

Figure 5.2 – Event communication architecture overview.

5.4 Correlation Engine Evaluation

As a core component of the platform, the correlation engine, had to be carefully chosen. This section describes the tests performed to the correlators as well as the features analyzed in order to select most suitable correlation engine, to be used in the correlation platform.

5.4.1 Performance Evaluation

As the correlation platform should process and correlate events in near real-time as well as detect attacks in the shortest time possible, a performance evaluation was conducted to the selected group of correlation software. The idea was to get an overview of their capabilities in terms of event processing speed and resources usage (like memory and CPU). Additionally the goal was to provide additional information that could assist in the selection of the software to be used as correlation engine. By having a hands-on, experience it was possible to have an insight of how easily was to express correlation rules in each of the correlators rule language, the configurability of the engines and eventual limitations.

As some tools were excluded to be considered as valid options for the correlation platform, as described in section 3.5, the tests were executed only for remaining tools: Drools, Esper, SEC and NodeBrain.

The test setup and results are detailed in the next sections.

5.4.1.1 Test Setup

The performance tests were executed on a virtual *CentOS* 6.4 operative system running on a server with the following characteristics: *Intel Xeon* CPU X5660 with 2.80GHz. The virtual system had only one single core allocated.

The tested versions of the software were: *Esper* 4.9, *Drools* 5.5.0, *SEC* 2.74 and *NodeBrain* 0.8.15. Both *Esper* and *Drools* used Java, the installed Java version was *OpenJDK* 1.7.0_25.

5.4.1.2 Tests and Results

The tests consisted in processing input events, lines of text in a *Syslog* similar format, read from a file by the correlation engine. With a defined number of correlation rules. The correlation rules were defined to check for income matching events using regular expressions. When an event was matched by a rule, the engine logged this occurrence to a log file. The correlation process logs allowed to certify that the different tools were matching the same events.

As all correlators use different rule language, the rules had to be written in such a way that they were equivalent between the different tools.

The tests were performed three times, and the results were averaged.

Correlators	**Events per second (for different number of input events)**											
	1,000 events			**10,000 events**			**100,000 events**			**1,000,000 events**		
	20 rules	**200 rules**	**500 rules**	**20 rules**	**200 rules**	**500 rules**	**20 rules**	**200 rules**	**500 rules**	**20 rules**	**200 rules**	**500 rules**
Drools	302	172	128	2,109	1,342	1,053	10,830	7,318	6,247	22,850	14,824	14,075
Esper	448	291	210	2,947	1,724	1,053	16,685	4,652	2,030	37,294	5,767	2,318
NodeBrain	6,383	4,347	3,571	6,756	4,511	3,726	6,678	4,494	3,744	6,352	4,329	3,611
SEC	3,225	735	322	4,231	4,231	345	4,443	1,559	345	4,400	811	338

Table 5.1 – Correlators throughput comparison.

From the tests, it could be observed that *Drools Fusion* was the software that consistently needed more memory to run the tests, followed by *Esper*. *SEC* and *NodeBrain* used, in almost all tests, nearly the same amount of memory. One of the reasons why *Drools* and *Esper* used more memory can be attributed to the fact they are tools written in Java and, therefore, additional memory is required to run the JVM. In Figure 5.3, is shown an example of the memory usage during the execution of a test with 100,000 input events and 500 rules. It should be referred that *Drools* could not finish tests with 1500 or more rules due to memory limits. However, the other tools had no problems finishing this tests. Tests containing rules a with counter for events in a sliding window were not possible to run with Drools, this engine crashed when processing events with this type of rules.

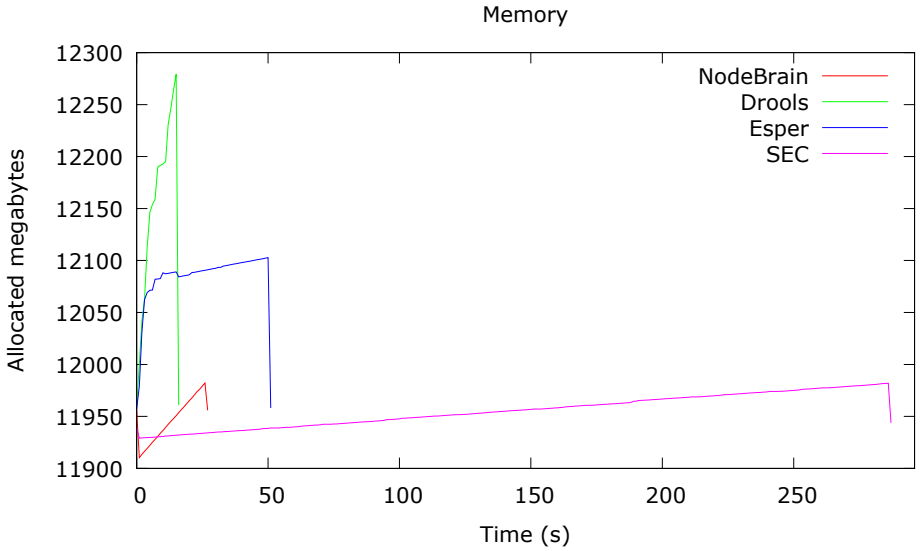

Figure 5.3 – Memory usage comparison, 500 independent rules with 100,000 input events.

The throughput results are presented in Table 5.1. From the results, it can be observed that the number of the input events has almost no impact in number of events processed by second, by *SEC* and *NodeBrain* while for *Esper* and *Drools* it can be seen that the throughout increases with more events. The fact that *Drools* and *Esper* have lower throughput than *SEC* and *NodeBrain*, for a small number of input events (1,000 and 10,000), seems to be related the impact of the time need to start the JVM, as this tests take less than 10 seconds to run. It should be noted that in real usage the impact of starting time of the JVM is negligible as the engines are going run continuously.

Increasing the number of rules has a larger impact in decreasing the throughput in SEC and Esper than in *Drools* or *NodeBrain*, but it is in *SEC* that increasing the number of rules has the biggest impact.

For tests with a large number of events and that run for longer (100,000 and 1,000,000) SEC has the lowest throughput results, while *Drools* has the best results, except for the small set of rules (20 rules). In this case, *Esper* had better results. However, such a small number of rules is not expected in a correlator in real usage.

It was observed, from the result data, that I/O operations, like reading the events and writing the log files, were never a bottleneck in the tests. In all tests the bottleneck was CPU bound, Figure 5.4 shows the CPU usage while processing 100,000 input events with 500 rules.

Although the presented tests does not represent a complete view of the correlators performance, they can give however an indication of the throughput expected by each one of the tools in different conditions.

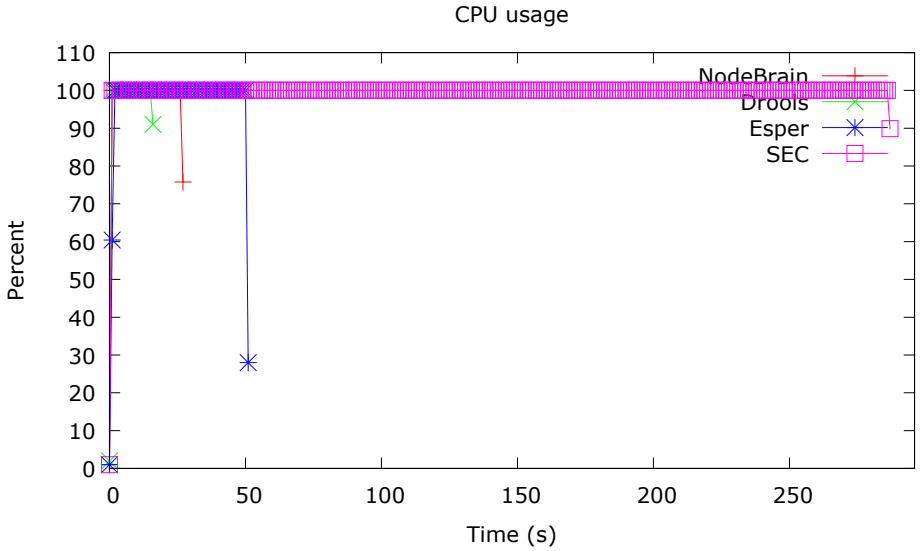

Figure 5.4 – CPU usage comparison, independent rules with 100,000 events.

5.4.1.3 Conclusions

Based on what was outlined above and the features provided by the tools, detailed in Table 3.1 and 3.2, *Esper* is the recommended correlator engine to be used by the security correlation platform of the PIDS.

The good performance exhibited by *Drools* and an easy to write rule language, close to natural language, could not overcome the problems exhibited by *Drools* during the tests. This engine could not complete some of the tests due to the lack of enough memory, even when the JVM memory limits were increased to values that were the double of those used by *Esper*. *Drools* also crashed when using a counter within a sliding window rule. In all tests, *Drools* was the tools that required more memory resources to run.

SEC proved to be the slowest correlator, at least in the executed tests. This and the fact that the rules that can be expressed by its language are less flexible, when compared to the other three tools, lead to consider the other tools first.

Regarding NodeBrain, even though it had good throughput results its rules proved to be very unintuitive to write and cumbersome. Moreover, this tool has almost no user community with almost no discussions on its mailing lists. Additionally, like SEC, this tool is designed to parse events from text lines using regular expressions. Therefore, it would be more complex to parse IDMEF XML events as the input events had to be converted to a format that could be understood by the tool.

Esper proved to be the right tool to be used as local and global correlator of the correlation platform. Although it was not the top performance correlator tool, it presented good results without using excessive memory. This correlation engine is also actively maintained with a very active user community. As pointed before, it can accept

input events in XML format that make it suitable to accept IDMEF events. Additionally, with a SQL like language, the rules proved to be intuitive to write. The expressivity of the Esper query language EPL, that is, the ability of the query language to express determined correlation scenarios was considered very good by [68].

Esper has some limitations in the open source versions, but they are mainly related with the lack of high availability and graphical user interface and not as a performance limitation.

Chapter 6

Implementation and Integration

This Chapter discusses the steps and the decisions taken during the implementation of the correlation platform, as well as the integration of existing intrusion detection software and other components with the platform. The main components implemented were the EventBus for event communication, the correlator application for event correlation and the agents that integrate the NIDS and the HIDS with the platform.

6.1 Event Communication: The EventBus

The EventBus or just Bus is used to transmit the events in the distributed system between the different components.

The communication uses the AMQP protocol, as defined in section 5.3.4.1. There are several broker implementations as well as client interfaces in different languages. Some of the most widespread are RabbitMQ [1], [2] and Apache Qpid [3].

The RabbitMQ server (version 3.2.0), was the broker selected to implement the EventBus. It was found that at a feature level there were no large differences among the different implementations, which would invalidate the choice of one of the implementations. The main reason to select the RabbitMQ was the fact that it provided more detailed and complete documentation. It had detailed tutorials for broker and clients API in different programming languages. As a result, it helped to support other external teams to connect and send events to the platform as well as to make the development easier. Some informal benchmarks showed that RabbitMQ was also one of the fastest broker.

[1] http://www.rabbitmq.com

[2] https://activemq.apache.org

[3] https://qpid.apache.org

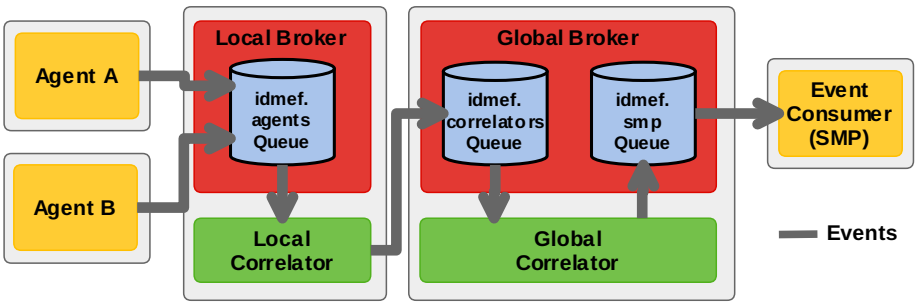

Figure 6.1 – EventBus and correlators configuration layout.

It should be noted that although the AMQP is RabbitMQ main protocol it also supports the MQTT, STOMP protocols.

The applications that connect with a broker or Bus were implemented using different client libraries as described in the next sections.

6.1.1 EventBus Configuration

As described in Section 5.3.4.2, there is one event broker deployed in each one of the PIDS network security zones: IT Network, Operations Network and Field Network. These brokers received the events published from the several in the corresponding zone that are then consumed by the local correlator in each zone. In the upper level there is a global broker that takes the responsibility for the events published by the local correlators and then consumed by the global correlator. After processing the events the global correlator publishes the events to a queue in this broker. The events published by the global correlator are sent a different queue from were they will be consumed by the Security Management Platform. The EventBus layout and correlators is presented in Figure 6.1. This diagram is simplified in a way that it only includes one Local correlator, to make it easier to describe. To represent the architecture described in Section 5.3.4.2 it is just a matter of adding one or more combos Local broker/Local correlator, where the Local correlator publishes to the *idmef.correlators* queue in the Global broker.

The local brokers have similar configurations, in these brokers a queue was configured ,named *idmef.agents,* binded to a *Topic* exchange (named *pids_exchange*) with the *idmef.agents* routing key. In a Topic exchange, an event sent with a particular routing key will be delivered to all the queues that are bound with the matching binding key.

The queues were configured as non auto-delete, durable queues. Queues configured this way are permanent and are not deleted by broker when there is no consumer or publisher interacting with it.

The global broker as two queues configured, one named *idmef.correlators* that is responsible for queueing the events sent by the Local correlators and from where the Global correlator consumes these events. The other queue named *idmef.smp* is to where the Global correlator publishes the events and from where the SMP will consume them.

6.1.2 Message Reliability

To provide message reliability to the event communication is was used the confirmation mechanism provided by *RabbitMQ* to guarantee that a message hasn't been lost (called *publisher confirms*) [69], this is an extension to AMQP that provides increased performance when compared to the transaction mechanism defined in AMQP for message reliability. Transaction's decrease throughput by a factor of 250, according to [69], which can be unacceptably slow. Using the *publisher confirms*, once a channel is put into confirm mode the broker will confirm messages as it processes them asynchronously, so the publisher does not need to wait for the broker to acknowledge the last message. The publisher just sends the messages and will receive asynchronously *an* acknowledge or a negative acknowledge from the broker, in case broker has acknowledge or not the message. The broker acknowledges a message when it assumes responsibility for it, can it be that the message was persisted to disk, or consumed from every queue it was sent. Instead, with transactions the publisher needs to wait for the broker to process the last message and receive an acknowledgment, due to its blocking nature.

6.1.3 Event Publisher Library

In order to facilitate the integration of the different security sensors and to be easily reusable, the EventBus publisher was implemented as a python library, the *eventbus* module. This library was used for publishing events to the Bus by the Honeypot, ShadowRTU, OSSEC Agent and Snort Agent. Figure 6.2 shows a high level diagram of the EventBus publisher.

The choice of python as the programming language was related to the fact that the Honeypot and the ShadowRTU are being developed in Python, therefore, this language allowed for a seamless integration with these applications. Moreover, the developer already had experience with this language, and there are several well supported libraries that implement the AMQP 0-9-1 protocol in python. It should be noted that the development of the ShadowRTU and Honeypot carried out by other members of the LCT CockpitCI team was started prior the development of the Distributed Correlation System.

The library selected for the connection with the RabbitMQ broker was the *pika* version 0.9.13 [70] python client library. *Pika* provides a pure python implementation of the AMQP 0-9-1 protocol supporting synchronous and asynchronous connections, and it is very well documented.

The goal of the development of this library was to provide a robust, reliable and easily integrable software to send messages to the EventBus.

The library is composed of the *EventBus* class and the *EventSender* class. The *EventSender* class implements the connection management logic as well as sending event to bus, while the *EventBus* class is the class integrated with the application that want to send the events to the bus. The *EventBus* class works as an abstraction for the application that makes use of the library. The application only has to instantiate the class, start it and call the method *send("Message text")* to send a message to the bus.

When stopping the application the stop() method must be called to stop in an orderly way.

The *EventSender* is spawned as different thread of the main application to avoid blocking. The communication of the events from the main application integrating the *EventBus* class and the EventSender is implemented using the *deque* class from the Python standard library module *collections* [71]. Deques are a generalization of stacks and queues that allow for memory efficient appends and pops from either side of the deque. This provides a caching mechanism for messages that cannot be send due to a lack of connection with the broker.

To avoid losing messages and in the event that there are unsent messages while stopping the application the *eventbus* module allows these messages to be persisted to disk. The messages can also be persisted to disk if the number of the messages in the memory buffer (*deque*) reaches a certain, configurable, threshold. This is to avoid holding a large number of messages in memory. It is particularly important when using the module in hosts with very limited resources like the single-board computers used by the SmartRTU and Honeypot.

A FIFO persistent priority queue was used to persist the messages to disk the messages while maintaining the order of the messages; the implementation of this type of queue was provided by the python module *queuelib*[4].

Three priorities (priority1, priority 2 and priority 3) were used in the persistent priority queue according to the source of the messages to persist. This allows the module internally to keep the order of the messages, it does not relate to the event content priority, sent events does not have priorities. When consuming from this queue, the messages with the highest priority are always consumed first (lower the number higher the priority). The priority assigned to each of messages according to their source is described bellow:

- Priority 1: messages sent to the Bus and waiting for acknowledgment or messages that received a negative-acknowledge. The communication of this messages from the *EventSender* to the *EventBus* is implemented by means of a *Queue* from the Python standard module *Queue*[5].
- Priority 2: messages in the memory buffer, waiting to be sent to the bus before stopping the application.
- Priority 3: messages that are persisted to disk when the memory buffer reached its maximum size, when the buffer has free space messages are moved from this queue to the memory buffer.

Messages persisted to disk will be sent to the bus upon application restart.

The *EventSender* class implements a connection recovery mechanism that reconnects the client to the broker in case of connection failure. If the broker is not

[4]https://pypi.python.org/pypi/queuelib

[5]https://docs.python.org/2/library/queue.html

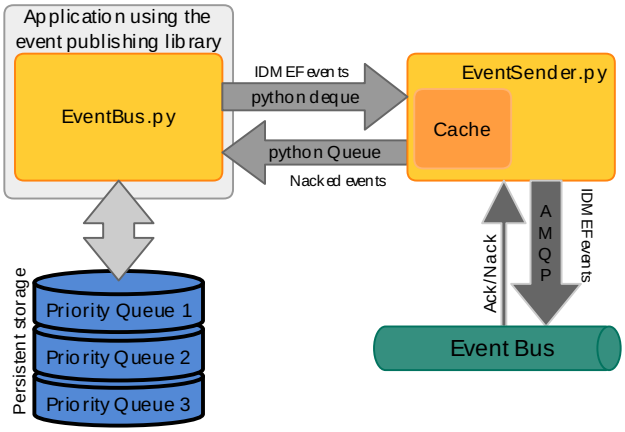

Figure 6.2 – Python EventBus publisher.

available, the module attempts to connect/reconnect. The interval between subsequent reconnection attempts is configurable.

The communication between the *EventSender* and the *RabbitMQ* broker is established by a permanent connection. Connecting every time a message needs to be sent to the broker increases the latency, the latency is even worst when using TLS encrypted connections due to the handshake step where the peers negotiate the cipher suite, establish the secret keys for the connection, and authenticate their identities.

TLS connections provide encryption, authentication, and data integrity. With this configuration, only clients with a valid signed certificate are allowed to connect to the broker and send messages to the Bus.

The confirmation mechanism provided by RabbitMQ is used in the *EventSender* class. The container *OrderedDict* (similar to a Python dictionary but it remembers the order that keys were first inserted), from the Python standard library module *collections* [72], is used as a cache to keep track of the messages sent to the bus. When an event is delivered to the Bus is is added to the container, when the acknowledge is received the event is removed from the container. If an negative-acknowledge is otherwise received the event is moved to another container. The messages in this container are resent again to the bus.

When stopping the application if there are unacknowledged messages and negative-acknowledge the containers with this messages are merged, sorted by message delivery, to be persisted to disk, as previously described.

Pika can use different connection adapters that allow it to use different I/O loop implementations for *pika* core communications. The *EventBus* is implemented using the *pika Tornado Connection Adapter*. This adapter uses the *Tornado ioloop* event loop. It is an I/O event loop for non-blocking sockets used by the *Tornado*[6], the Python web framework and asynchronous networking library. This loop is used because it allows to use the same I/O loop for listening to messages that come from the application through

[6]http://www.tornadoweb.org/en/stable/

the python *deque* and consume them, as well has sending the messages to the Bus and receiving the broker acknowledges. Otherwise, two different I/O event loops needed to be used one to interface with the python *deque* and other with the Bus.

6.2 Event Correlation

The correlation application has three main components, the input adapter, the correlation core and the output adapter. In this section, is described how the implementation of these elements was performed.

The application was implemented using the Esper engine Java API; this engine was selected as correlator according to the reasons described in 5.4.

As described in Section 5.4.1.3 the same application can run in Local and Global correlator modes. The correlator mode defined at the application start by passing a command line parameter, passing *-global* or *-local* according to the desired mode. The distinction between these two types of correlator applications are the input and output adapter configurations, Esper engine configurations, correlation rules as well as the listeners (Java classes that process the output of the rules) that can be associated to the rules.

In Figure 6.3 is presented the diagram of the implemented correlator application, the diagram presents a more detailed view of the correlator than architecture overview presented in Figure 5.1.

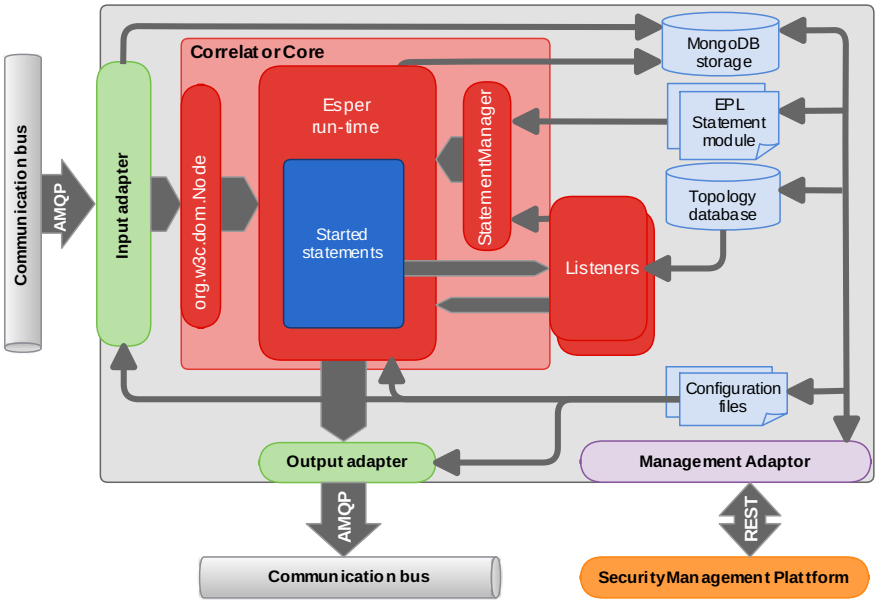

Figure 6.3 – Correlator design diagram.

6.2.1 Correlator Core

The correlator core component is responsible for the event correlation, statement management, life-cycle management of the input, output and management adapters as well as managing the configuration of the application.

In the context of the correlation engine platform, a correlation rule can be composed of one or more EPL statements.

To understand how the correlator core application was implemented around the Esper engine is important to describe the Esper Architecture. A simplified overview of the Esper engine architecture can be found in the Figure 6.4

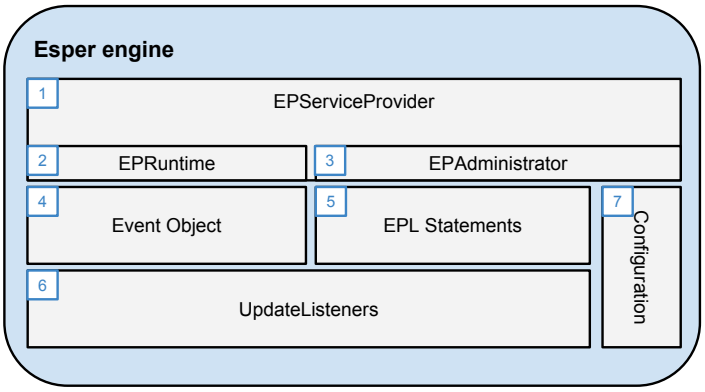

Figure 6.4 – Esper engine architecture overview.

1. The *EPServiceProvider* interface. This interface represents an engine instance. Each instance of an Esper engine is completely independent of other engine instances and has its own administrative *EPAdministrator* and runtime *EPRuntime* interfaces.

2. The *EPRuntime* interface is mainly used to send events for processing into an Esper engine.

3. *EPAdministrator* is used to create pattern expressions, create EPL statements, receive statement results and other statement operations. The results can be obtained by attaching one or more listeners to a statement, the EPAdministrator interface also provides this.

4. Event object, which is an object that represents an event. Esper can accept events in different formats, like POJO, Java Maps, XML and Java Object arrays. The statement results events are returned as *EventBean* objects. These objects are events that have additional metadata associated. The metadata differs for each of one of the event formats.

5. The *EPLStatement* are queries written in EPL. Statements are the correlation rules in other correlator's terminology.

6. The *UpdateListener*, these listeners are Java classes that receive updated data as soon as the statement processes the incoming events. The listeners attached to statements receive *EventBean* objects that represent a row (event) in a continuous query's result set. An *UpdateListener* implementation receives one or more *EventBean* events with each invocation.

7. Configuration, to tune the engine to specific configurations an application using the engine can use XML files to hold the configuration as well as using the Configuration class at the time of engine allocation. The configuration can also be changed at run-time using the *EPAdministrator interface.*

Although Esper can have several engine instances running at the same time, in the implemented application only one engine is instantiated. As the application needs be aware and process all the events entering the correlator, and all events of the same type and representation, there were no advantages of instantiating more than one instance.

Bellow is presented a description of how the features of the correlator core application were implemented:

Configuration to simplify the deployment of the correlator, as it can run in Local correlator or Global correlator modes, the configuration files for the input adapter, output adapter, statements module and Esper engine, have distinct paths for the Local and Global modes. When the correlator starts, it loads the configurations concerning the chosen mode.

As pointed before, Esper can manage different event representation and have different types, the event type describes the type of information for an event representation. Predefined event types can be configured at the application start or at run-time via API or EPL statements. The events entering the engine via input adapter are in the IDMEF format (XML based), represented as *org.w3c.dom.Node* instances. An XML event representation for the type IDMEF was configured in Esper including the required XML root element name, for the IDMEF the root node is *"IDMEF-Message".* The IDMEF XSD (XML Schema Definition) file, included in the RFC [62], was set in the configuration. This allows Esper to validate EPL statements that refer to event properties against the types provided in the schema. An example of the configuration can be found in the Appendix C.

The option *xpath-property-expr* was enabled to allow the traversal of the namespace-aware Document Object Model (DOM) representation of the IDMEF. With this option, the engine rewrites each property expression as an *XPath* expression to access the event properties. Property expressions are expressions in EPL statements that allow to assess event properties. For example, the property expression Alert.Classification.text, for the configured IDMEF event, allows to get the value of the attribute *text* defined in the element *Classification,* child of the IDMEF element *Alert.* Internally this expression is rewritten as the equivalent *Xpath* expression */idmef:IDMEF-Message/idmef:Alert/idmef:Classification/@text*, including the *idmef* namespace prefix, that to query the intended XML node value.

Explicit properties, defined with the option *xpath-property*, can be configured to access some IDMEF properties stored in child elements instead of attributes, additionally this can provide shorter properties expressions to access the most often used properties. These are properties explicitly defined in the configuration, there is a property name which backed by a Xpath expression.

Statement management the statements, representing the correlation rules are read and parsed from a file upon the engine initialization. The file is loaded as an EPL module. In *Esper*'s terminology, an EPL module is a plain text file in which EPL statements appear separated by a semicolon (;) character. After added to the list of known modules, it is deployed starting all the statements of the module. When undeploying the module all started statements associated to the module are destroyed. All the module management is done throughout the *EPDeploymentAdmin* service available from the *EPAdministrator interface.*

To allow for the management of the statements at run-time, without the need for restating the correlator, a file change monitor was implemented using the *WatchService* interface from the standard *java.nio* package present in Java 1.7 [73]. This service watches registered objects for changes and events. The file with the statements is registered with the service, to be monitored for file changes. When the statements file changes, a callback is called to the reload the module.

Every time the module is loaded, the implemented application validates it to check whether the included set of statements is complete and can start without issues. This is accomplished by deploying the module to an isolated service provider. An isolated service is an execution environment separate from the main run-time engine. This is done with the option to not deploy any EPL statement and just only perform syntax checking. If there is a syntax error on the statement, an error is logged appending the error description provided by *Esper*. Most of the times the messages provided by Esper are descriptive enough to identify the error including the line number where it can be found. If there are syntax errors in the module, it is not deployed, only after changing the file and correcting the error(s) the module is deployed to the running engine instance.

One of the challenges, while developing, the correlator was to define a way to dynamically attach a listener to a statement as well as to add new listeners without the need to compile to byte-code the entire correlator application every time a new statement or listener was added/removed or changed.

As Esper only allows to add a listener class to a statement programmatically through its API, the solution was to make use of Esper annotations. An annotation is additional information added to the statement; it is part of the statement text and precedes it. They can be used, for example to define, the statement name *using @Name("example name")*, allowing to retrieve the statement later on by the engine, or add a statement description using *@Description("example event description")*.

Esper provides certain built-in annotations, but applications can provide their own annotation classes that the EPL compiler can populate. Therefore a new annotation class was created, the *@UpdateListeners* annotation, with the purpose off attaching

one or more listeners to a statement from the module file dynamically at run-time. A list or a single *UpdateListener* class can be attached to a statement be appending the *@UpdateListeners* annotation before the statement. The syntax for adding a list of listeners to a statement is as follows: @UpdateListeners({"ListenerClassA", “ListenerClassB”}).

When the module is loaded, the application will use a factory method to dynamically load and instantiate the classes with the names matching the ones identified in the *@UpdateListeners* annotation. The factory method will look for the classes matching the name in a defined path and will raise an exception if a class with that name cannot be found. As a result, it is possible to move a pre-compiled listener to the defined path and attach it to a statement during run-time, without the need to restart the correlator application. Additionally with this feature it is easy the Local and Global correlators to have different listeners without the need to recompile the application. The *StatementManager* class, identified in Figure 6.3, implements the functionalities described above.

Listeners Listeners are Java classes that receive the results of the statements, they can receive the results as *org.w3c.dom.Node* instances or Java Map instances. These classes, after processing the events are responsible for building the new IDMEF events, when required, and can also execute other programmed actions. To build new IDMEF messages in Java, it was used the *JavaIDMEF*[7] library. As this library only supported an old version of the IDMEF standard, it had to be modified to support the latest version.

6.2.2 Input Adapter

This adapter is responsible for consuming events from the Bus and insert them into the *Esper* engine for correlation. It interfaces with the Bus using the RabbitMQ Java client library to consume events from the broker.

The adapter was implemented as an *Esper* plug-in implementing the *com.espertech .esper.plugin.PluginLoader* interface from the Esper API. By implementing the input adapter as plug-in as the advantage that the plug-in follows the Esper engine life-cycle; when the engine initializes, it instantiates the PluginLoader implementation class. The engine then invokes the lifecycle methods of the *PluginLoader* implementation class before the engine is fully initialized (*init* method, where is done some variable initialization) and after (*postInitialize* method, where is established a connection to the broker) and before the engine instance is destroyed (*destroy* method, where the connection to the broker is terminated).

The *ConsumerManager* implements the connection management to the broker with connection recovery. When the connection to the broker is not explicitly terminated by the application, this class implements a recovery mechanism that reconnects to the broker. If the broker is not available it, keeps trying to reconnect.

The TLS protocol used to provide encryption, authentication, and data integrity to the connection.

[7] http://sourceforge.net/projects/javaidmef/

All the connection parameters are read from a file upon initialization.

The *ConsumerWorker* class is responsible for the processing of the events consumed from the broker. As an event arrives from the Bus as a XML string (representing the IDMEF), it is converted to *org.w3c.dom.Node* instance, this interface represents the DOM an entire XML event, which the Esper engine can process. After the conversion, the event is sent into the engine via the *sendEvent* method on *the* Esper *EPRuntime in*terface. If the event is insert correctly into the engine an acknowledge is sent to the broker, to inform it that the correlator has taken responsibility for the event. Otherwise, a negative-acknowledge is sent so the broker can re-queue the event unless the exception is an *Esper EPException*. In this case, the event is acknowledged, and the error logged. As this indicates an Esper run-time exception, the event should not be re-queued because if there is a problem with the event format it would be negative-acknowledged every time.

All the events processed by the *ConsumerWorker* are logged to a MongoDB[8] database for auditing purposes.

6.2.3 Output Adapter

Instead of implementing a new output adapter as a plug-in, like it was done for the input adapter, the Esper *AMQPSink*[9] data-flow operator was used. While for the input adapter there were advantages of implementing a new adapter as a plug-in, as this provided more flexibility and control of the event consumer, that could not be achieved with the built-in Esper input adapter. For the output adapter, there was a feature that could not be provided by a plug-in. The *AMQPSink* data-flow operator allows for an event to be sent to the Bus directly from an EPL statement without requiring a listener class to process the event. This is useful when forwarding events to the upper layer, as it that does not require further processing.

To the correlator publish an event to the Bus, using the AMQPSink, this event must be wrapped in new event type, configured as *OutgoingWrappedIDMEF and* represented by a Java Map containing the IDMEF XML string. Every time an event of this type enters the Esper engine, it is published to the Bus.

The *AMQPSink* code was modified and extended to provide additional features, such as connection recovery, TLS protocol support, connection parameters configuration read from an external file, confirmation mechanism for the messages and message caching. Like the input adapter, the modified adapter supports reconnection to the broker upon disconnection, encryption and authentication via TLS. The confirmation mechanism guarantees that a message is not lost, as described in Section 6.1.2. Message caching allows to store the messages in memory when there is a connection failure, the messages will be published to the Bus when the connection is re-established.

[8] http://www.mongodb.org/

[9] http://esper.codehaus.org/esperio-5.0.0/doc/reference/en-US/html/adapter_amqp.html#amqp-sink

6.3 Agent integration

In this section is described who the integration of the NIDS and HIDS was implemented. These intrusion detection tools are one of the sources of the security events that provide information to be correlated.

6.3.1 NIDS Integration: The Snort Agent

The Snort Agent is responsible for reading Snort events, compose IDMEF messages with the data obtained from the event and send the event to the EventBus.

To read the *Snort* logs it was used the *idstools* [10] tool, a python library to work with snort rules and logs. This library allows for continuous *unified2* directory spool reading, including bookmarking support. The bookmarking feature allows the library to bookmark the last *unified2* log file, as well as the last event in the file, this allows the reader to remember its location and start reading from the bookmarked location on initialization, avoiding parsing all the previously read events. Each event processed by the tools is represented as a python dictionary dict containing the fields of a *Snort unified2* event record.

In order to use this library, *Snort* had to be configured to log with the *unified2* output plug-in. The unified2 is designed to be the fastest possible method of logging *Snort* events. According to the *Snort* documentation [74], this output plug-in logs events in binary format, allowing other programs to handle complex logging mechanisms that would, otherwise, diminish the performance of *Snort*.

The agent runs as a daemon, when a new *Snort* event is logged to the *unified2* log file the daemon composes an IDMEF file with the information from the *Snort* event and sends the event to the EventBus using the event publishing library, described in Section 6.1.3.

The agent has an event reduction mechanism. An event is filtered if the same event is generated from snort during a certain interval; this interval is configurable.

Two events are considered equal if they have the same hash. The hash is obtained from hashing the concatenation of the words (strings) of the defined event properties, like signature id, destination IP, source IP, protocol, priority, etc. The list of the properties to include in the string to hash is configurable. This allows to configure how aggressively the events are filtered. When fewer properties are selected to be included results in more events considered equal. Hence, more events are going to be be filtered. So, this properties need to be chosen carefully. Event timestamp, for example, should not be used as a propriety for hashing as, otherwise, no events could be considered equal.

For hashing it was used a fast non-cryptographic hash, Murmur3[11] or optionally xxHash[12] were selected. It is used the hashing library available in the platform were the

[10] https://pypi.python.org/pypi/idstools

[11] https://code.google.com/p/pyfasthash/

[12] https://github.com/ewencp/pyhashxx

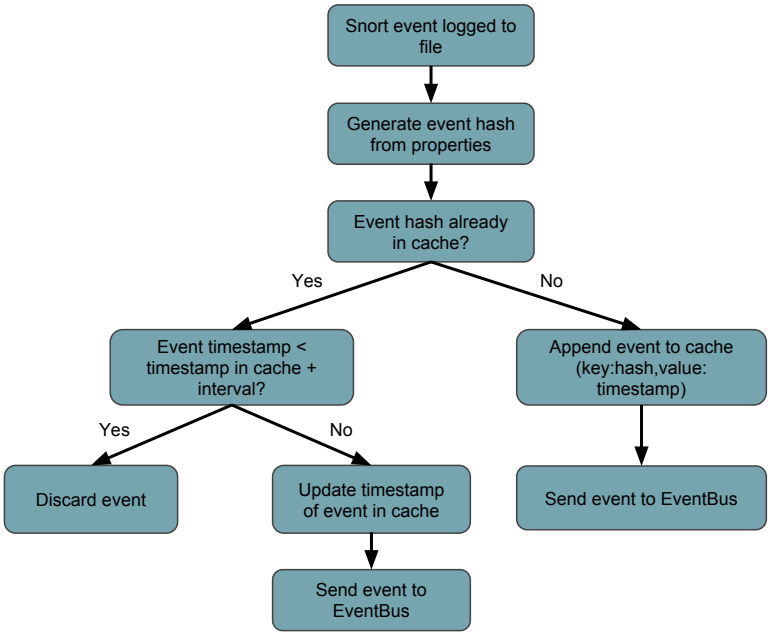

Figure 6.5 – Snort agent event filtering flow

agent is going to be deployed. These fast hashing functions have good performance and have a low number of collisions.

A python dictionary is used to keep a temporary cache of previous events. The events are inserted into this dictionary with the event hash as the key and the timestamp as the value, the timestamp represents the time of insertion into the cache. Upon arrival of a new event, the hash is generated for the event, as described above, and it is inserted into the cache with the current time as timestamp. If an event with identical hash is found to be already in the cache upon insertion the timestamp of this event is verified. If the number of seconds that elapsed since the time represented by the timestamp of the event in cache is less than a defined (configurable) interval of time the event is filtered and not sent to the EventBus. Otherwise, the timestamp is updated with the current time. The event flow described above can be pictured in Figure 6.5.

As there could be a large number of unique event hashes, for long periods of execution, the cache could became very large. To prevent the cache to keep growing indefinitely entries with expired timestamps need to be removed. To avoid iterating all the elements of the dictionary for an expired timestamp, on a regular interval or on every new insertion, an operation that for large caches could delay the processing of a new event, a different approach was used. Upon new event arrival only a set of events in cache is checked for expiration, the events are removed from cache if found expired, an event is considered expired if more than a configured number of seconds have elapsed since the value of timestamp for that event. The size of the set is configurable and the set is chosen randomly from all the keys in the dictionary. It was found that with set with 1 or 2 elements to be enough to avoid the cache to grow excessively, with larger sets

the overhead would be larger without reducing the size of expired events in cache.

To build IDMEF events, programmatically in a structured way, it was used the *GenerateDS*[13] python application. This application is an automated tool that generates Python data structures from a XSD schema. This tool generated a python module, the *idmef_api.py*. However, the module had to be adjusted to produced valid IDMEF files as there where some details of the IDMEF that were not correctly generated by the tool.

6.3.2 HIDS Integration: The OSSEC Agent

The *OSSEC* architecture includes a central manager for monitoring and receiving information from agents. The agents are small programs installed on systems to be monitored, that are registered within the manager.

OSSEC provides several ways of sending alerts to other systems or applications, such as send alerts via e-mail, Syslog and SQL database.

To integrate the *OSSEC* output with the correlation platform was implemented using the Syslog output from the *OSSEC* manager. This option was found to be the simplest and with the lowest overhead. It had the advantage of using a service already running on the host, *Rsyslog*, to forward the messages without that need of more complex daemon to parse e-mail or SQL database.

The integration of the syslog with the EventBus was implemented by using the *omprog*[14] module (Program integration Output module) from the *Rsyslog*[15], a Syslog daemon implementation. This module allows to integrate external an external program with the Syslog daemon. For this, an external application was developed. This application, *omprog_ossec.py*, receives from standard input (*stdin*) the Syslog messages the *omprog* module outputs. *Rsyslog* was configured to send all messages coming from *OSSEC* to this module, in JSON (JavaScript Object Notation) format. To send messages to the Bus, it was used the event publisher library, referred in Section 6.1.3. Before sending the messages to the Bus, the application assembles the events in IDMEF format from the information contained in the JSON message received from the *Rsyslog* daemon.

6.3.3 Other Agents and Systems Integration

As pointed before, the Shadow RTU and the Honeypot, developed by other elements of the CockpitCI LCT team, were integrated with the correlation platform by using the python event publisher library described in Section6.1.3.

Additionally, is was provided support to integrate security tools developed by other CockpitCI project members. One example is the multi anti-virus software checker developed by itrust[16]. A security company from Luxemburg. Their tool successfully

[13]https://pypi.python.org/pypi/generateDS

[14]http://www.rsyslog.com/doc/omprog.html

[15]http://www.rsyslog.com/

[16]http://www.itrust.lu/

connected to the correlation platform. This tool publishes IDMEF events to the platform reporting alarms from the hosts being analyzed.

Another tool is the OCSVM (One-Class Support Vector Machine) module, a machine learning tool for intrusion detection developed the University of Surrey[17], in the U.K. This module analyzes the network traffic connects to the correlation platform and publishes reports in the IDMEF event format.

[17] http://www.surrey.ac.uk/

Chapter 7

Validation

This Chapter discusses how the developed application validation was performed. Two types of validation were performed, functional validation and performance validation.

7.1 Functional Validation

In this section are described the validation steps performed to the correlation platform in order to verify that is was working according to the requirements. For the validation in this section it was used a simplified small scale test bed, running in a virtualized server. A an overview of the configuration is depicted in Figure 6.1.

7.1.1 Preliminary validation

A preliminary and simplified version of the correlation application and EventBus was demonstrated in the Conference on Innovation for Secure and Efficient Transmission Grids[1] (CIGRE), that was held on Brussels, Belgium from the 12th of march to 14 of march of 2014.

The scenarios demonstrated in the conference showed the capabilities of CockpitCI detection layer to detected, correlate and report some security attacks. The attacks were communicated to the Integrated Risk Predictor (IRP), described in 2.1. The IRP was developed by Roma Tre University, from Italy.

The following scenarios were successfully tested: Network Scan, Network Flood, Honeypot Interaction and Man-in-the-Middle attack.

All the CockpitCI team collaborated in the development of the demo. The author of this thesis was in charge of the event communication and correlation tasks.

[1] http://www.cigre.org/Events/Other-CIGRE-Events/Innovation-for-secure-and-efficient-transmission-grids

7.1.2 Correlation

The correlation application needs some rules configured to perform the event correlation. In the section are described some correlation operations applied to the events received in the correlator. The statements described were written to the application EPL statement module file, loaded by the application. Several publisher agents published a set of events composing a scenario that triggered the correlation operation. This allowed to verify if the outcome of the correlation operation was the expected. These tests also allowed to verify the statement management functionality, as the statements were configured while the correlator was running. The statements or rules described here show the capabilities of the platform to perform security event correlation, with IDMEF events. Additionally it should function as a guide on how to write rules for the correlation application. The goal is not to show all the event processing capabilities provided by *Esper* neither to test all possible correlation operations.

7.1.2.1 Event Aggregation

In this section is described an example of an event correlation operation named event aggregation. This operation consists in creating a new event, that has a new meaning, from a set of events. The aggregated event contains references for the events it aggregates.

The example implements an Event Storm detection. Event storms are the manifestation of an important class of abnormal behaviors in communication systems, according to [75]. They occur when a single host generates an excessive number of events within a small period. It is essential for network management systems to detect every event storm and identify its cause, in order to prevent and repair potential system faults.

The statements that allow detecting an event storm can be found in Listing D.1 in Appendix D. The output of the statement is sent to a listener, this listener then builds an IDMEF event with results from the *EventStorm* statement. The generated IDMEF is then inserted into the Esper engine to be sent to the upper correlation level by the correlator output adapter.

The example statements in Listing D.1 allow to detect when a single host, generates more than 100 events during an interval of 120 seconds. As one event IDMEF can have multiple sources, an event stream, which aggregates all the sources, is first created. An event stream is a time ordered sequence of events in time. The events in this stream have only a subset of all the properties of the IDMEF from where they originate. This stream can be seen, in a simplified way, as the list of sources of all events entering the engine.

When the correlator is configured with the statements presented in Listing D.1 and more than 100 events, identifying the same address as source, arrive at the correlator within an interval of 120 seconds an IDMEF alerting for an event storm is sent to the correlator output adapter. The generated IDMEF contains the reference for each of the event ids (*messageid* attribute in the IDMEF) that originated the event storm, as well as the target's IP addresses of the events. It should be noted, that a listener

IDMEFEventStormListener, developed specifically for this rule, need to be attached to the last statement in Listing D.1 in Appendix D, so it can receive the results of the statement.

To test the rule, several events containing different sources each were sent to the correlator application. It was observed that when one source IP was referred 100 times in the received events during the defined interval, it was generated and sent to the output adapter an IDMEF containing the references for those events.

7.1.2.2 Event Filtering

The statements in Listing D.3, found in Appendix D, were configured in the correlator application to provide event filtering. With the configured statements the correlator filters all but the first event, every 30 seconds, originated from any *Snort* Agent and reporting a SYN Flood. This type of events is received in large batches of similar events from the Snort Agent. As a result, the correlator only forwards to the upper layer, the first of these events, every 30 seconds, filtering all the other similar events.

7.1.2.3 Event Suppression

The event suppression statements, presented in Listing D.2, found in Appendix D, have a similar operation than the ones presented before, i.e. both make event filtering. However, while the event filtering presented before took into account properties of the events to be filtered, the event filtering in Listing D.2 also take into account the state of the correlator. In this particular case, the *"ARP Cache Overwrite Attack"* event from *Snort* is only forwarded to the output adapter if a Medium or Severe Alarm has been received from the OCSVM in the previous 5 minutes, otherwise the event is filtered. An *Esper* variable is used keep the alert level from OCSVM. The variable is toggled by statements that check for events originated from the OCSVM.

7.1.3 Resilience

To test the resilience of the application some tests were performed. Some failures were intentionally triggered, to verify how the components recover for failure while events are being published by the agents. Below are described the tests performed. In the tested configuration it was not implemented broker or correlator redundancy mechanisms. However this configuration can be implemented, which can provide additional resiliency and robustness to the platform. The tested setup diagram is pictured in Figure 6.1.

Broker failure: the local broker was rebooted during normal operation. The idea was to simulate a broker failure. It was observed that while the broker was down the publishing agents kept trying to connect to the broker, as well as, the input adapter of the local correlator. The events produced by Agent A and B were queued by the event publishing library while the connection was down. When the broker started again,

the agents successfully reconnect to it and sent all unsent cached events. The global correlator reconnected again to the broker and started consuming the events.

The same procedure, as described before, was performed for the global broker. It was observed that the local correlator output adapter cached the events not sent while the connection with the global broker was down. As soon as the global broker was operational the output adapter was able to reconnected to it, and sent the cached events. Both input adapter and output adapter kept trying to reconnect to global broker while it was down, once it restarted both adapters successfully reconnect to it.

During this procedure, no events were lost due to the implemented caching mechanisms in the agents and local correlator output adapter.

Correlator failure: the local correlator was restarted while operating. When the correlator was down the events published by the agents were queued in the broker. When the correlator started, connected to the broker and started consuming the queued events. When executing the same operation with the global correlator, the events received by the local correlator were queued in the *idmef.correlators* in the global broker. As soon as the global correlator restarted both connection adapters reconnected to broker and started processing the events.

In this test there was event loss as *Esper* does not persist the events when stopping, so the events the *Esper* engine acknowledged and were being processed were lost (*Esper* does not support event persistence in the Open Source version). The events eventually queued in correlator output adapter are also lost as it was not implemented persistence for this caching mechanism.

Agent shutdown: testing the correlator resilience was done with the local broker shutdown, in this situation it was observed that since the agent publishing library was unable to connect to the broker the events, received from the application using the library, were being cached by the library. The application was then restarted. When stopping the publishing library persisted the cached events to disk. Once agent application restarted it started trying to connect to the broker again, as soon as the broker was started, it was observed that the cached events were successfully sent to the broker.

Consumer failure: When the consumer was shutdown while operating, the events were cached in the *idmef.smp* queue. As the confirm mechanism, previously described in Section 6.1.2, was enabled, the events that were not acknowledged by the consumer were republished by the broker. No events were lost during this failure.

Although only single component failure at the same time was tested, it was possible to verify the platform provides strong resiliency capabilities.

To provide more reliability to the platform is also possible to have more than one *RabbitMQ* broker connected in a cluster, and using High Available queues, as described in [76]. In this configuration, the queues are mirrored, and the events in them synchronized.

The publishers and consumer connect to the cluster if one broker node fails the other takes its place.

Having redundancy mechanism on the correlators (be it Local or Global) allows the consumer to start receiving events from the queue with events from the backup correlator. Both share the same internal state as in this configuration the master and slave correlator are receiving the same events and have the same configured rules. This configuration protects from a failure of both the correlator and the broker at the same time. The configuration previously described is presented in the Figure 7.1. It should be taken in consideration that this solution was empirically tested, no further testing was done as it was out of the scope of this thesis.

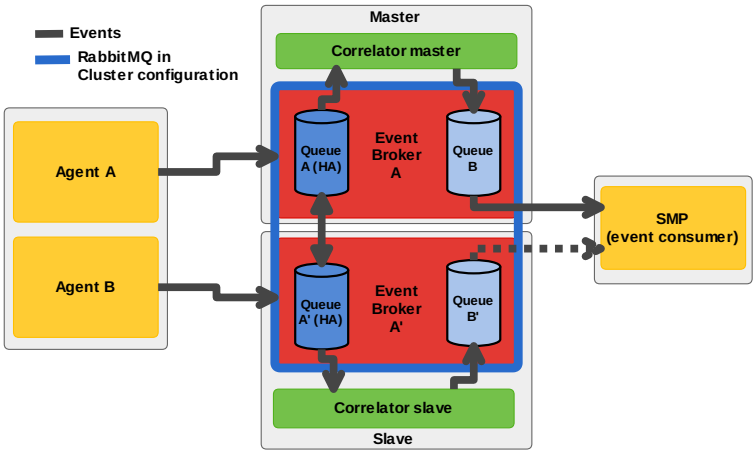

Figure 7.1 – High availability correlation platform configuration.

7.2 Performance testing

To assess the performance of the correlation platform, some benchmarks were performed. The results of these tests can allow to identify eventual bottlenecks in the platform, estimate the performance behavior in a working deployment as well as define what elements can be optimized.

7.2.1 Event Publishing on Limited Resources Systems

The performance of an event publisher in a small low-spec ARM computer, like the *Raspberry Pi* [77], was also tested. The purpose was to check the feasibility of an AMQP event publisher on this type of system and assess the performance. This type of computer will be used to deploy the Shadow RTU and Honeypot. Therefore, this components need to send security events to the to the Bus. The testing was successful, and it was possible in a simple test to send to the broker events (in IDMEF format with 1Kb) at a rate of around 200 events/second, on average.

7.2.2 EventBus and Correlation Application Test Setup

The benchmark layout was configured like shown in Figure 7.2. The correlation platform was deployed in three different hosts. Host 1 and host 2 identified in the Figure were deployed on a virtual *CentOS* 6.4 operative system running on a server with the following characteristics: *Intel Xeon CPU X5660* with 2.80GHz. The virtual host had only one single allocated. These hosts have the same configuration than the system used for the correlator benchmarks referred in Section5.4.1.1. Host 3 was not a virtualized system; it was a system running outside the server. An event publisher was deployed in host 1 (simulating an agent); this element publishes events in IDMEF format to the broker using the Python eventbus library described in Section 6.1.3. In host 2, it was deployed the RabbitMQ broker and the correlation application, while host 3 had an application that consumed the events from the broker.

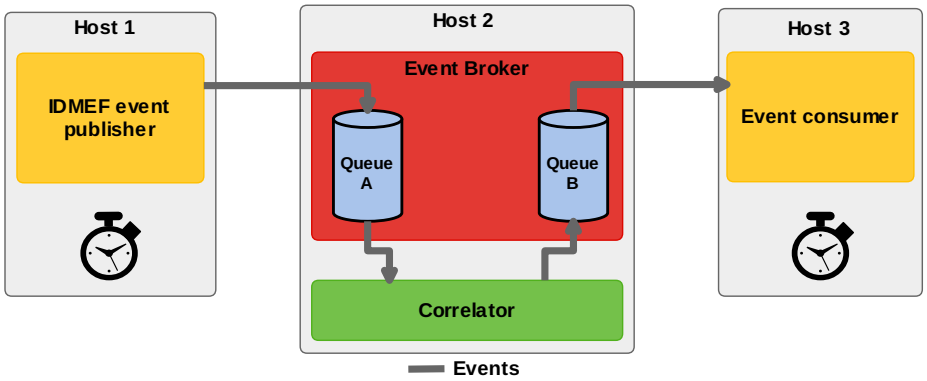

Figure 7.2 – Performance testing configuration layout.

To time the tests the publisher writes the current system time on one element of the IDMEF then when the consumer receives the event it reads the publisher time and compares it to the arrival time. Both machines system clock's were synchronized with a Network Time Protocol (NTP) before performing the benchmarks.

The correlation application and the RabbitMQ broker were deployed in the same host as in this configuration the communication between the two components can happen inside the same machine without increased network latency, the correlator consumes events from the broker and then after processing publishes the to the broker. Additionally, as communication takes place inside the same host, the encryption of the communication between the two components can be disabled, reducing the overhead encryption imposes without lowering the security level.

It should be noted, that the testing configuration represents a segment of the architecture described in Section 5.1. As the architecture includes two correlators the latency values should be doubled, as proposed architecture includes two correlators it is equivalent of having two testing layouts connected in sequence, like publisher $\rightarrow$ broker$\leftrightarrow$correlator $\rightarrow$ broker$\leftrightarrow$correlator $\rightarrow$ consumer. This simplified testing layout allows to identify the eventual bottlenecks better.

The correlator was configured with 20 rules. The tests in this Section were

focused in the global platform performance assessment. Detailed benchmarks of the correlator component are described in Section 5.4.1.2. In this performance testing, the correlator was not filtering events, neither new events were created by the correlator. All and only the events consumed were being published to the broker. When the configured rules were triggered, by matching a configured string with an element in the IDMEF event, the events were published to Queue B. All events were triggering a rule.

7.2.3 Event Rate Evaluation

Several tests were performed to test the number of events the correlation platform can process when publishing events as fast as possible.

IDMEF events do not have a size limit and can have different sizes to accommodate information from different sources, for example, an IDMEF that is the result of the correlation of several other events includes the references for all the events it aggregates as well as their sources and targets, thus becoming a very large event. As a result, to test the impact of the message size in the message rate, the tests were performed for different message sizes, from 1 kB to 20 kB. The 1 kB represents an IDMEF with all the mandatory fields and 20 kB an estimation of an event with several references for other events. However, an IDMEF event can even be much larger than 20 kB tested here.

In order to have an idea of the impact in the performance imposed by confirmation mechanism, for improved reliability, described in Section 6.1.2 the tests were also performed with this mechanism disabled. When enabled, an acknowledge is sent to the publisher when the consumer processes the message or the broker persists the event to disk. Otherwise, when disabled, in auto acknowledge mode (*AutoAck* as RabbitMQ names it), the broker sends an acknowledge to the publisher as soon it receives the event.

Although the encrypted communications are a requirement, the tests were also performed without encryption to assess how it impacts the performance of the platform. The values presented are the average of the execution of each test three times.

All the tests were executed with messaged marked as persistent; this way messages are persisted to disk, by the broker, when they can not be immediately consumed. This allows the events to have higher delivery guarantees, however, this also decreases the performance.

Figure 7.3 shows the impact of the message size in the message rate when sending 10.000 events to the broker, as fast as possible.

The results show, as expected, that when the message size increases the rate of messages consumed decreases, as larger messages take longer to send and process. It can be seen that the impact of the confirmation mechanism and encryption is not neglectable and that this impact, in the message, rate decreases with the size of the message. For event sizes of 5 kB to 20 kB, the rate reduction is primarily due to the confirmation mechanism, as encrypted and non-encrypted rate can be considered very similar.

In the Figure 7.4 are shown the results from the same tests, but this time, comparing the number of sent bytes versus the message size. As it can be verified, when increasing the event size, the number of bytes sent increases while rate of events decreases.

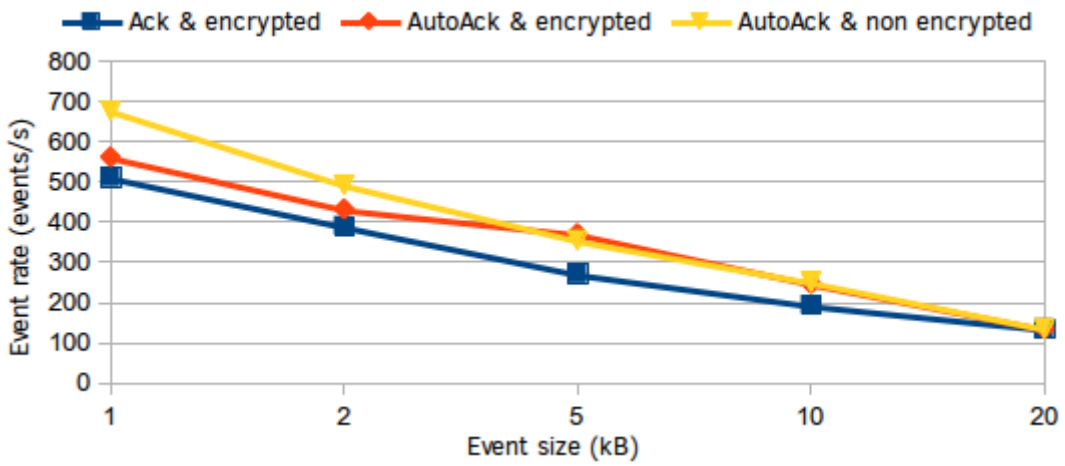

Figure 7.3 – Event rate for different message sizes.

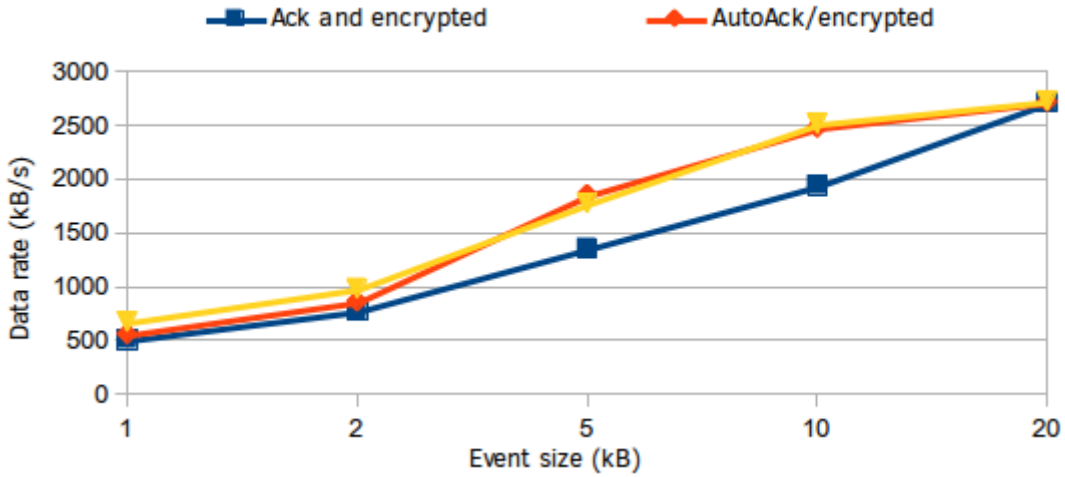

Figure 7.4 – Event data rate for different event sizes.

The CPU usage for the Host 2, where the correlator and the broker were running, is shown in Figure 7.5. As it can be seen is reached the maximum capacity of the CPU. The process that took most of the CPU time was the correlator, taking on around 70% of CPU time.

While the event publisher was sending the events, Host 1 CPU was also at 100% of maximum CPU capacity. Host 3 never peaked 100% of CPU usage.

The system memory never reached its limit in any of the Hosts.

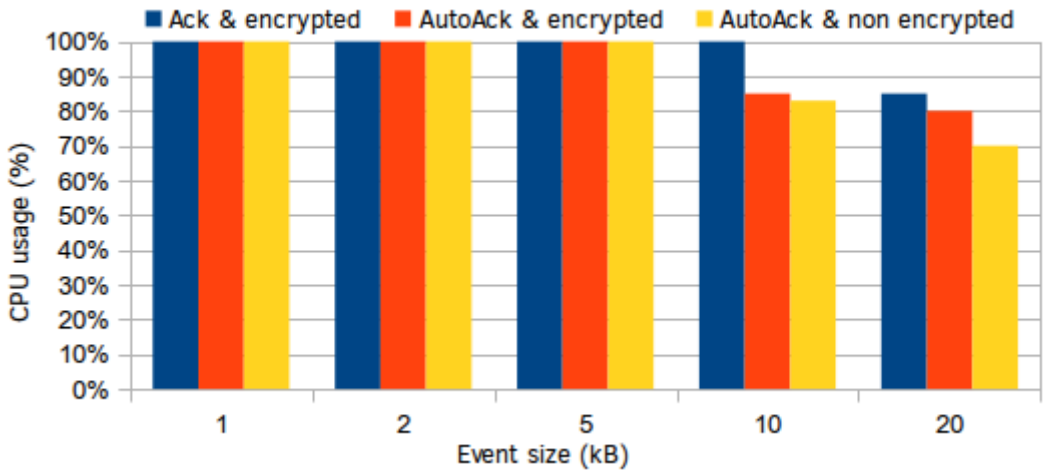

Figure 7.5 – Host 2 CPU usage during tests.

Another point that should be highlighted is that the event rate consumed, from Queue A, by the correlator application was always much higher than the rate the was publishing to Queue B, when the CPU usage reached 100%. It the worst case, the correlator in the worst case was only publishing at a rate of 30% of the consumed event rate. When the CPU usage was not reaching the peak, the correlator was publishing events with a rate equal at the rate it was consuming the events or only at a slightly lower rate. These values were observed in the graphical user interface of the broker while monitoring the tests; it was not possible to get this values a systematic way to do further analysis.

7.2.4 Latency Evaluation

When performing the test referred in the previous Section the average one-way latency was also measured, it represents the latency from all sent events, from when they are sent until they are consumed. They results show very large average latency values when sending the 10.000 messages, as fast as possible, for messages sizes lower than 10 kB or 20 kB, depending on the transmission confirmation method and if encryption is enable or not. As can be seen in Figure 7.6, the latency is in the order of seconds when the CPU of Host 2 reaches 100%. This show that when the CPU reach its maximum capacity, due to the processing of the events by the correlator and event broker, the average latency changes from the milliseconds range to the seconds range. With the CPU at 100% the event broker and correlator have to wait for processing time and as result the events will take longer to process. When the CPU usage of host 2 does not reaches its peak, for larger event sizes, as they take longer to be sent by the publisher and are received more gradually, the latency ranges, on average for all 10.000 events, from 200 ms to 300 ms. Additionally, the events that are not immediately consumed by the correlator are persisted to disk by the broker what also justifies the high latency values. It was observed that not using auto acknowledging the events has a large impact on the average

latency, except for events with 20 Kb.

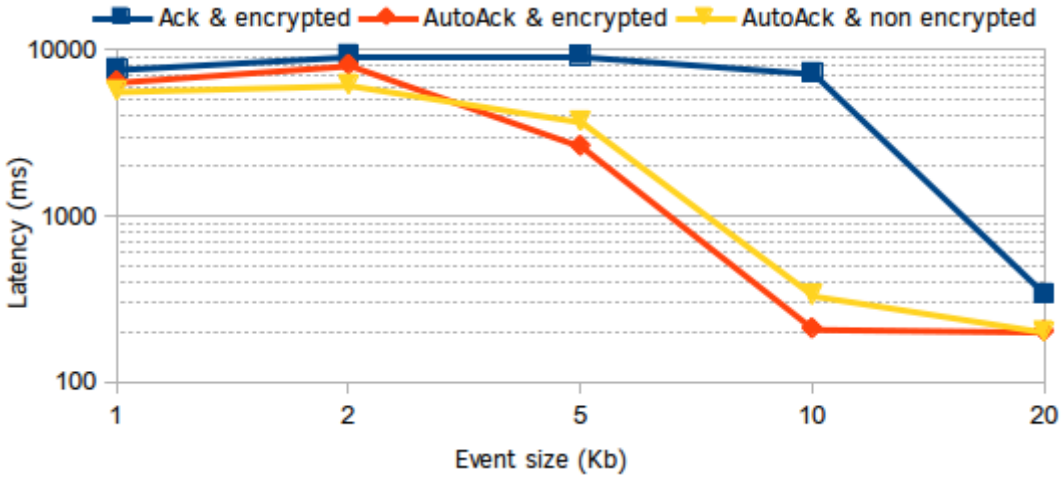

Figure 7.6 – Average latency for different event sizes.

The tests previously described demonstrated that when Host 2 reached its maximum processing capabilities the latency increased considerably. Taking this into account, additional test were performed. In this tests, the number of consecutive messages sent to the broker continuously. The results are present in the graph of Figure 7.7. The tests were executed for events of 1 kB and 5 kB.

The chart in Figure 7.7 shows that for the testing deployment when the number of events, sent continuously, was over 1000 the latency became extremely high as the Host 2 reached its maximum capacity. When the system receives a lower number of continuous events, the latency was below 1 second.

These tests showed that for a number of consecutive events that do not overload the Host 2 CPU, the impact of encryption and manual confirmation mechanism, although noticeable, does not have a large impact on latency.

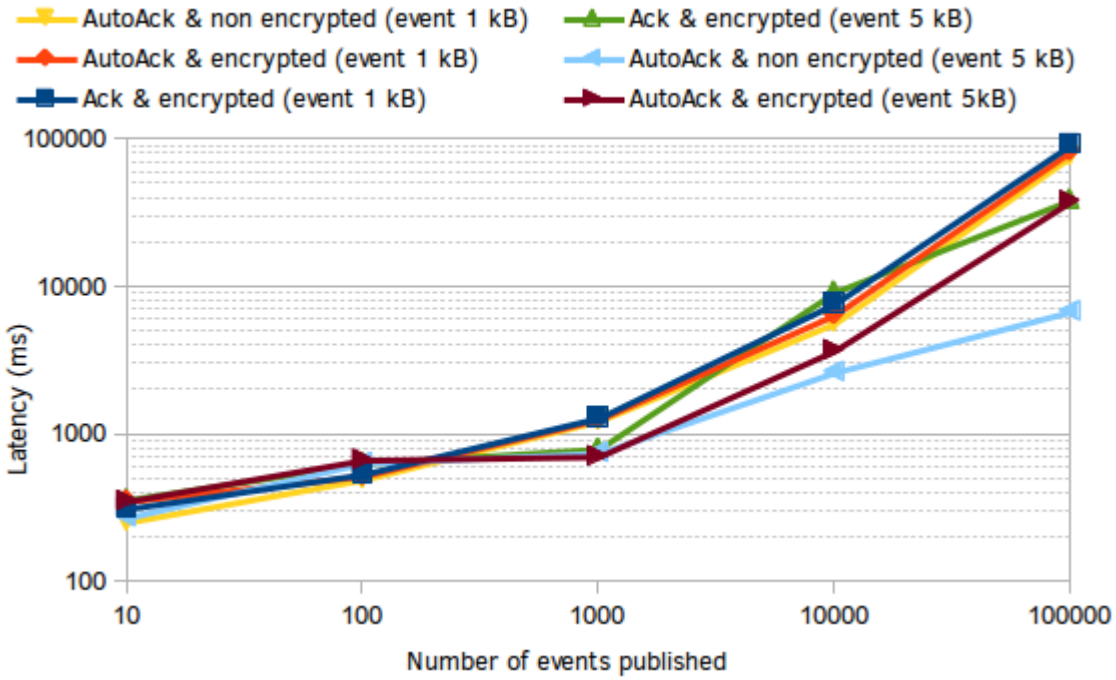

Figure 7.7 – Average latency for different number of events published continuously.

7.2.5 Performance Testing Conclusion

The tests showed that the platform can process a considerable number of events per second in a system with limited resources, as a single core virtualized system, where the tests were performed.

It was shown that when the CPU of the processing host does not reach its CPU usage limits the latency can be kept below the 1 second. Although this cannot be considered a low value, it should be highlighted, that the tests were performed publishing the batch of events in a continuous loop. This is the test that stresses the event broker and mainly the correlator.

Based on the observed values between the rated of events consumed by the correlator and the rate of the events it published after correlation it proved to be a bottleneck.

There are several aspects that can be taken in consideration when trying to improve the performance of correlation platform in terms of event throughput and latency:

- The host where the correlation application and the *RabbitMQ* were deployed had only a single core. This made the correlation application and the event broker, two resource hogging applications, impacted each other when the CPU reached its peak. A multi-core system would allow to decrease, if not eliminate, this impact in terms of performance.

- The flexibility of the platform allows the deploy the broker and the correlation application in different systems. In this configuration, the applications don't

system share system resources, although this can increase the latency slightly when transmitting the events between the two systems.

- As pointed before, several *RabbitMQ* brokers (nodes) can deployed in a cluster configuration (the brokers be deployed in different hosts), distributing the load among the different nodes. It is even possible to add new nodes without stopping the broker.

- *RabbitMQ* has several performance related parameters that can be tuned to eventually improve the performance. Esper provides as well several threading and concurrency parameters that can be adjusted to obtain better performance.

- The correlator application was built using one engine instance, in order to the correlator to receive all the events therefor has a global overview of all events. However is possible to increase the number of Esper engines instances. Based on the results of the correlators performance assessment, described in Section 5.4.1.2, when the number of rules increases the number of events processed by the correlator decreases. Having more than one Esper engine instance, where all the instances receive all the events, but each one with a different set of rules, it is possible to reduce the impact of of the number of rules in the correlation application performance. This can provide better scalability to the platform.

- The publisher library was developed using the python *pika* library and the publisher host reached the maximum CPU usage while publishing the events. Possibly using other library, as the Java client library) it would allow to increase the rate of events published for the same host. However, this would require capable broker/correlator.

It should be remarked that during in the tests all the events published reached the consumer, there were no lost events, even when the machine reached its processing limits, and the events were being queued.

While the evaluation performed in this Section stressed the correlation platform with a large batch of continuous events, when this platform will be deployed in the CockpitCI testbed, the number of continuously published events is expected to be lower then the limits tested here. Most of the agents will employ filtering mechanism that will reduce the event of the events published. The correlator will all be configured with rules for event filtering, aggregation and event suppression that will reduce the number of the events published to the upper layer.

Chapter 8

Project Progress

This Chapter discusses the work progress of the tasks performed in the second semester.

8.1 Constraints

There was a constraint in the course of the work for this thesis that changed the planning of some tasks. Due to some problems in the deployment of Hybrid Test Bed (HTB), beyond the control of the UC LCT team, the test bed was not available as defined schedule in the CockpitCI schedule. Only on the on July of 2014 the first machines were accessible from Coimbra Laboratory.

The test bed is located in the premises of the Israel Electric Corporation[1] (IEC), another partner of the CockpiCI project.

This constraint delayed the deployment of the platform to the test bed that would have provided a better and complete testing and validation of the platform. The platform was instead deployed into a small scale test bed in server located in the LCT laboratory.

This limited, in a certain way, the testing and validation presented in this thesis. The work that was allocated for a more extensive testing, validation and attack definition was either allocated for a more complete implementation of the Snort and OSSEC Agent implementation, as well as, improving the resiliency of the different components of the platform.

[1] www.iec.co.il

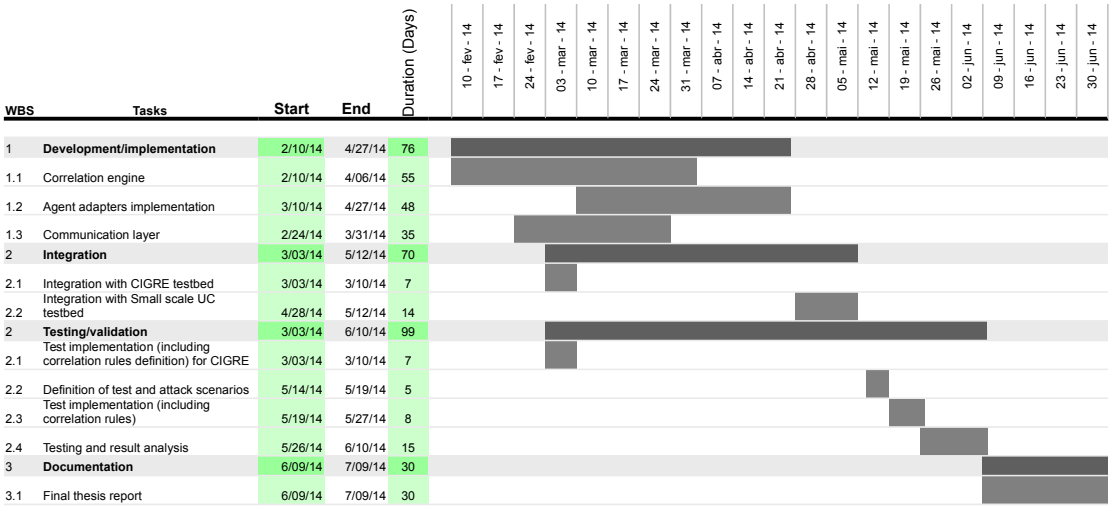

Figure 8.2 – Work progress progress Gantt chart.

8.2 Second Semester Work Progress

The Gantt chart in Figure8.1 details the work plan for the second semester, defined in the intermediate report. In the Figure 8.2 is presented the Gantt chart for main tasks performed during the second semester. The deviation for the scheduled plan and the plan executed was mainly due to the constraints referred in the previous section.

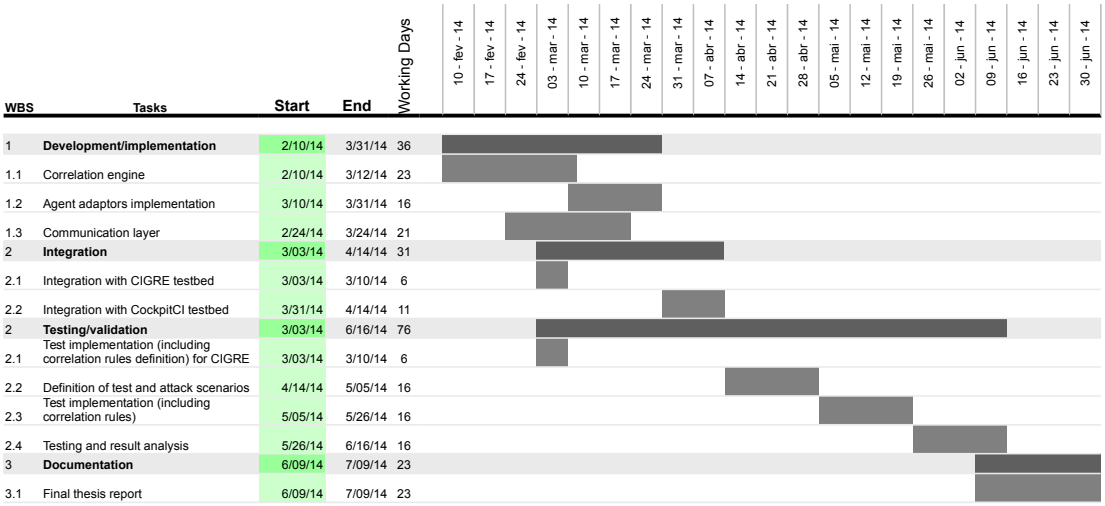

Figure 8.1 – Work planning Gantt chart.

Chapter 9

Conclusions

This document presented the architecture, implementation and validation of a distributed security event correlation platform.

The designed platform provides the capabilities to allow the PIDS to collect and correlate security events form distributed probes and varied IDS.

Beyond the correlation capabilities, the evaluation showed that the platform provides a tight integration of the intrusion detection systems with the correlation system. Therefore, allowing the collection and transmission of security events, from a multitude of security sensors distributed across several hosts in SCADA network to central a correlation system, in a secure and reliable way.

The architecture of the platform outlined here can be configured with rules that allow the PIDS to detect intrusions or intrusion attempts by extracting more useful information from the security events collected by the several sensors, highlighting the most important ones and reducing the number of false alarms. By integrating, as its core *Esper*, a complete CEP engine with a very expressive language assures that complex correlation scenarios can be configured.

The evaluation also showed that the implemented platform can provide high levels of resiliency. This is a vital property in a system designed to process security events. In this context, it is of great importance to be able to recover quickly from failures to avoid losing events.

The interoperability provided by using a standard message format and standard wire-protocol was demonstrated by connecting several components from different teams, using different Operative Systems and with applications written in varied programming languages. Hence, allowing, in the future, for more detection probes to easily be integrated with the platform, providing further information to the correlation application.

Although, the two-level correlation already provides a scalable solution, it was showed that the scalability by can be further increased by using a broker in cluster configurations in combination with additional correlator instances when the number of

rules increases at a level that impacts the performance of the correlation application.

In the next sections are presented other contributions in the scope of the project, as well as, future work to be developed.

9.1 Contributions

Based on the research and development performed in this thesis, the author of this thesis made several contributions to following deliverables of the CockpitCI project:

- Deliverable D3.1 - Requirements and Reference Architecture of the Analysis and Detection Layer (co-author and co-editor);
- Deliverable D3.2 - Real Time Intrusion Detection Strategies (co-author);
- Deliverable D3.3 - Design of Detection Agents and Field Adaptors (co-author and co-editor);
- Deliverable D3.4 - Design of the Dynamic Perimeter Intrusion Detection System (co-author and co-editor);
- Deliverable D3.5 - Implementation and Trials (co-author);

The author of this thesis had an active role on the definition of the requirements and configuration of the Hybrid Test Bed (HTB), in order to assure that it provided the required means for the validation of correlation platform. This test bed includes different critical infrastructures simulated by real equipment and enterprise Industrial Control Systems.

Additionally, co-authored a scientific poster to be presented at the 13th European Conference on Cyber Warfare and Security (ECCWS), entitled "A Survey of Signature-based Event Correlators".

There is also an ongoing effort to produce a paper, in collaboration with other member of the team, about event correlators survey and their performance evaluation.

9.2 Future work

The worked developed in this thesis will be continued. Future work will be focused in the integration of the correlation platform with the Security Management Platform. The component to be developed will allow the Secure Management Platform to consume the events generated by the correlation platform, and after processing, sending them to the Security Mediation Network, as defined in section 2.1.

Another task that will be carried out will be the deployment of the platform to the HTB. This test bed will allow a more comprehensive testing, validation and integration of the correlation platform with other CockpitCI components. Furthermore, with the deployment on the test bed it will be possible to test the platform with more

realistic and complex attack scenarios and network infrastructure, as there will be real equipment and the PIDS will have a complete deployment.

Additionally, with the availability of the HTB and further definition of complex attack scenarios more correlation rules need to be developed.

Bibliography

[1] T. Cruz, P. Simoes, J. Proenca, Pedro Alves, Luis Rosa, Jorge Barrigas, M. Curado, E. Monteiro, E. Ciancamerla, A. Di Pietro, M. Minichino, S. Palmieri, M. Ouedraogo, C. Feltus, D. Khadraoui, and A. Graziano, "CockpitCI cyber-security on SCADA: risk prediction, analysis and reaction tools for critical infrastructures, d3.1.2 - requirements and reference architecture of the analysis and detection layer," Jul. 2013.

[2] F. Valeur, "Real-time intrusion detection alert correlation," Ph.D. dissertation, University of California at Santa Barbara, 2006.

[3] P. Teufl, U. Payer, and R. Fellner, "Event correlation on the basis of activation patterns," in *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Feb. 2010, pp. 631–640.

[4] Antonio Spadaro, "Event correlation for detecting advanced multi-stage cyber-attacks," Master Thesis, Delft University of Technology, Delft, 2013. [Online]. Available: http://www.tbm.tudelft.nl/fileadmin/Faculteit/TBM/Over_de_Faculteit/Afdelingen/Afdeling_Infrastructure_Systems_and_Services/Sectie_Informatie_en_Communicatie_Technologie/medewerkers/jan_van_den_berg/news/doc/A.Spadaro_Thesis-truly-final.pdf

[5] C. Krügel, T. Toth, and C. Kerer, "Decentralized event correlation for intrusion detection," in *Information Security and Cryptology - ICISC 2001*, ser. Lecture Notes in Computer Science, K. Kim, Ed. Springer Berlin Heidelberg, Jan. 2002, no. 2288, pp. 114–131. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-45861-1_10

[6] H. T. Elshoush and I. M. Osman, "An improved framework for intrusion alert correlation," in *Proceedings of the World Congress on Engineering*, vol. 1, 2012.

[7] S. A. Mirheidari, S. Arshad, and R. Jalili, "Alert correlation algorithms: A survey and taxonomy," in *Cyberspace Safety and Security*. Springer, 2013, pp. 183–197. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-03584-0_14

[8] D. Gorton, "Extending intrusion detection with alert correlation and intrusion tolerance," Ph.D. dissertation, Chalmers tekniska högsk., 2003.

[9] F. Pouget and M. Dacier, "Alert correlation: Review of the state of the art," Eurecom, Tech. Rep. EURECOM+1271, 2003. [Online]. Available: http://www.eurecom.fr/publication/1271

[10] Andreas Muller, "Event correlation engine," Master Thesis, Swiss Federal Institute of Technology Zurich, Zurich, 2009. [Online]. Available: ftp://ftp.tik.ee.ethz.ch/pub/students/2009-FS/MA-2009-01.pdf

[11] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "A coding approach to event correlation," in *Integrated Network Management IV*, ser. IFIP - The International Federation for Information Processing, A. S. Sethi, Y. Raynaud, and F. Faure-Vincent, Eds. Springer US, Jan. 1995, pp. 266–277. [Online]. Available: http://link.springer.com/chapter/10.1007/978-0-387-34890-2_24

[12] L. Lewis, "A case-based reasoning approach to the management of faults in communications networks," in *Ninth Conference on Artificial Intelligence for Applications, 1993. Proceedings*, 1993, pp. 114–120.

[13] Xinzhou Qin, "A probabilistic-based framework for INFOSEC alert correlation," Ph.D. dissertation, Georgia Institute of Technology, Jul. 2005. [Online]. Available: https://smartech.gatech.edu/handle/1853/7278

[14] G. Jakobson and M. Weissman, "Alarm correlation," *Netwrk. Mag. of Global Internetwkg.*, vol. 7, no. 6, pp. 52–59, Nov. 1993. [Online]. Available: http://dx.doi.org/10.1109/65.244794

[15] Esper-Tutorial, "Esper - tutorial." [Online]. Available: http://esper.codehaus.org/tutorials/tutorial/tutorial.html

[16] Esper-Website, "EsperTech - products - esper." [Online]. Available: http://www.espertech.com/products/esper.php

[17] Esper-FAQ-Website, "Esper - complex event processing FAQ." [Online]. Available: http://esper.codehaus.org/tutorials/faq_esper/faq.html

[18] Esper Project, "Esper developer mailing list." [Online]. Available: http://markmail.org/list/org.codehaus.esper.dev

[19] ——, "Esper user Mailing list." [Online]. Available: http://markmail.org/list/org.codehaus.esper.user

[20] NodeBrain Project, "NodeBrain open source project." [Online]. Available: http://nodebrain.sourceforge.net/index.html

[21] NodeBrain Modules, "NodeBrain node modules." [Online]. Available: http://nodebrain.sourceforge.net/package/nb/version/0.8/modules.html

[22] NodeBrain Project, "NodeBrain tutorial," May 2013. [Online]. Available: http://nodebrain.sourceforge.net/package/nb/version/0.8/release/0.8.15/nbTutorial/nbTutorial.pdf

[23] ——, "Nodebrain users mailing list." [Online]. Available: http://sourceforge.net/mailarchive/forum.php?forum_name=nodebrain-users

[24] SEC Project, "SEC - open source and platform independent event correlation tool." [Online]. Available: http://simple-evcorr.sourceforge.net/

[25] Risto Vaarandi, "SEC - a lightweight event correlation tool," in *Proceedings of the 2002 IEEE Workshop on IP Operations and Management*, 2002, pp. 111–115.

[26] J. P. Rouillard, "Real-time log file analysis using the simple event correlator (SEC)," in *Proceedings of the 18th USENIX Conference on System Administration*, ser. LISA '04. Berkeley, CA, USA: USENIX Association, 2004, pp. 133–150. [Online]. Available: http://dl.acm.org/citation.cfm?id=1052676.1052694

[27] SEC Project, "Simple event correlator (SEC) manpage." [Online]. Available: http://simple-evcorr.sourceforge.net/man.html

[28] SEC Tutorial, "Working with SEC- the simple event correlator." [Online]. Available: http://simple-evcorr.sourceforge.net/SEC-tutorial/article.html

[29] SEC Project, "SEC user mailing list." [Online]. Available: https://lists.sourceforge.net/lists/listinfo/simple-evcorr-users/

[30] Drools, "Drools - the business logic integration platform." [Online]. Available: https://www.jboss.org/drools/

[31] Drools Fusion, "Drools fusion." [Online]. Available: https://www.jboss.org/drools/drools-fusion

[32] G. Oguz, "Decision tree learning for drools," Master Thesis, Ecole Polytechnique Federale de Lausanne, 2008. [Online]. Available: http://infoscience.epfl.ch/record/126292/files/oguz-thesis_final.pdf

[33] N. Wulff and D. Sottara, "Fuzzy reasoning with a rete-OO rule engine," in *Proceedings of the 2009 International Symposium on Rule Interchange and Applications*, ser. RuleML 09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 337–344. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04985-9_31

[34] "Rete algorithm," Dec. 2013, page Version ID: 586897354. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Rete_algorithm&oldid=586897354

[35] MVEL, "MVEL - home." [Online]. Available: http://mvel.codehaus.org/

[36] Drools Fusion CEP, "Chapter 8-complex event processing." [Online]. Available: https://docs.jboss.org/drools/release/6.0.1.Final/drools-docs/html/DroolsComplexEventProcessingChapter.html

[37] Drools Fusion, "Drools user mailing list." [Online]. Available: https://lists.jboss.org/mailman/listinfo/rules-users

[38] Alienvault-OSSIM-Project, "OSSIM: Open source SIEM & open threat exchange projects." [Online]. Available: http://www.alienvault.com/open-threat-exchange/projects

[39] AlienVault, "SIEM solutions & IT security risk assessment tools by AlienVault." [Online]. Available: http://www.alienvault.com/

[40] Prelude Project, "Prelude-IDS prelude universal open-source SIEM project." [Online]. Available: https://www.prelude-ids.org/

[41] Prelude-IDS, "Prelude-IDS - prelude components." [Online]. Available: https://www.prelude-ids.org/wiki/prelude/PreludeComponents

[42] ——, "Prelude-IDS - PreludeCorrelator." [Online]. Available: https://www.prelude-ids.org/wiki/prelude/PreludeCorrelator

[43] ——, "Prelude-IDS - prewikka." [Online]. Available: https://www.prelude-ids.org/wiki/prelude/ManualPrewikka

[44] Quadrant Information Security, "The sagan log analysis & correlation engine." [Online]. Available: http://sagan.quadrantsec.com/

[45] Sourcefire, "Snort - open source intrusion and detection system." [Online]. Available: http://www.snort.org/

[46] ACARM-Project, "ACARM | main / home." [Online]. Available: http://www.acarm.wcss.wroc.pl/

[47] AlienVault, "AlienVAult forum: ossim open source version can supports multi-level deployment?" [Online]. Available: http://forums.alienvault.com/discussion/195/ossim-open-source-version-can-supports-multi-level-deployment

[48] Prelude-Project, "www.prelude-ids.org mailing lists." [Online]. Available: https://www.prelude-ids.org/lists/listinfo

[49] ACARM-ng-Project, "ACARM-ng / discussion / forums." [Online]. Available: http://sourceforge.net/p/acarmng/discussion/

[50] D. Menasce, "MOM vs. RPC: communication models for distributed applications," *Internet Computing, IEEE*, vol. 9, no. 2, pp. 90–93, Apr. 2005.

[51] Q. H. Mahmoud, *Middleware for communications*. Chichester, England: J. Wiley & Sons, 2004.

[52] G. Banavar, T. Chandra, R. Strom, and D. Sturman, "A case for message oriented middleware," in *Distributed Computing*, ser. Lecture Notes in Computer Science, P. Jayanti, Ed. Springer Berlin Heidelberg, Jan. 1999, vol. 1693, pp. 1–17. [Online]. Available: http://dx.doi.org/10.1007/3-540-48169-9_1

[53] H. Subramoni, G. Marsh, S. Narravula, Ping Lai, and D. Panda, "Design and evaluation of benchmarks for financial applications using advanced message queuing protocol (AMQP) over InfiniBand," *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on*, pp. 1–8, Nov. 2008.

[54] STOMP, "STOMP - the simple text oriented messaging protocol," 2014. [Online]. Available: https://stomp.github.io/

[55] J. Oraskari, "The performance of open message-oriented middleware protocols in smart space access," 2010.

[56] vmware, "Choosing your messaging protocol: AMQP, MQTT, or STOMP | VMware vFabric blog - VMware blogs," 2013. [Online]. Available: http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html

[57] MQTT, "MQTT," 2014. [Online]. Available: http://mqtt.org/

[58] StormMQ, "WhitePaper - a comparison of AMQP and MQTT," 2012. [Online]. Available: https://lists.oasis-open.org/archives/amqp/201202/msg00086/StormMQ_WhitePaper_-_A_Comparison_of_AMQP_and_MQTT.pdf

[59] Oracle, "Java message service (JMS)," 2014. [Online]. Available: http://docs.oracle.com/cd/B14099_19/web.1012/b14012/jms.htm

[60] AMQP Webpage, "AMQP about." [Online]. Available: http://www.amqp.org/about/what

[61] ——, "AMQP 1.0 becomes OASIS standard | AMQP." [Online]. Available: http://www.amqp.org/node/102

[62] IETF-Network Working Group, "The intrusion detection message exchange format (IDMEF) RFC." [Online]. Available: http://www.ietf.org/rfc/rfc4765.txt

[63] LibIDMEF, "LibIDMEF web site." [Online]. Available: http://sourceforge.net/projects/libidmef/

[64] OASIS, "OASIS advanced message queuing protocol (AMQP) version 1.0, part 5: Security." [Online]. Available: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-security-v1.0-os.html

[65] Apache-Qpid, "ClusteringHA - apache qpid - apache software foundation." [Online]. Available: https://cwiki.apache.org/confluence/display/qpid/ClusteringHA

[66] Pivotal-RabbitMQ, "RabbitMQ - clustering guide." [Online]. Available: http://www.rabbitmq.com/clustering.html

[67] Apache-ActiveMQ, "Apache ActiveMQ clustering." [Online]. Available: https://activemq.apache.org/clustering.html

[68] H.-L. Bui, "Survey and comparison of event query languages using practical examples," Ph.D. dissertation, 2009.

[69] RabbitMQ, "RabbitMQ - confirms (aka publisher acknowledgements)," 2014. [Online]. Available: https://www.rabbitmq.com/confirms.html

[70] "Pika documentation." [Online]. Available: http://pika.readthedocs.org/en/latest/index.html#

[71] python documentation, "8.3. collections - high-performance container datatypes - python v2.7.7 documentation," 2014. [Online]. Available: https://docs.python.org/2/library/collections.html#collections.deque

[72] Python documentation, "8.3. collections - high-performance container datatypes - python v2.7.7 documentation," 2014. [Online]. Available: https://docs.python.org/2/library/collections.html#collections.OrderedDict

[73] Javadocs, "WatchService (java platform SE 7)," 2014. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/nio/file/WatchService.html

[74] Sourcefire, "Snort - unified output plugin documentation," 2014. [Online]. Available: http://manual.snort.org/node249.html

[75] M. Albaghdadi, B. Briley, and M. Evens, "Event storm detection and identification in communication systems," *Reliability Engineering & System Safety*, vol. 91, no. 5, pp. 602 – 613, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095183200500116X

[76] RabbitMQ, "RabbitMQ - highly available queues," 2014. [Online]. Available: https://www.rabbitmq.com/ha.html

[77] Raspberry-Pi, "Raspberry pi | an ARM GNU/linux box." [Online]. Available: http://www.raspberrypi.org/

Appendix A

IDMEF data model

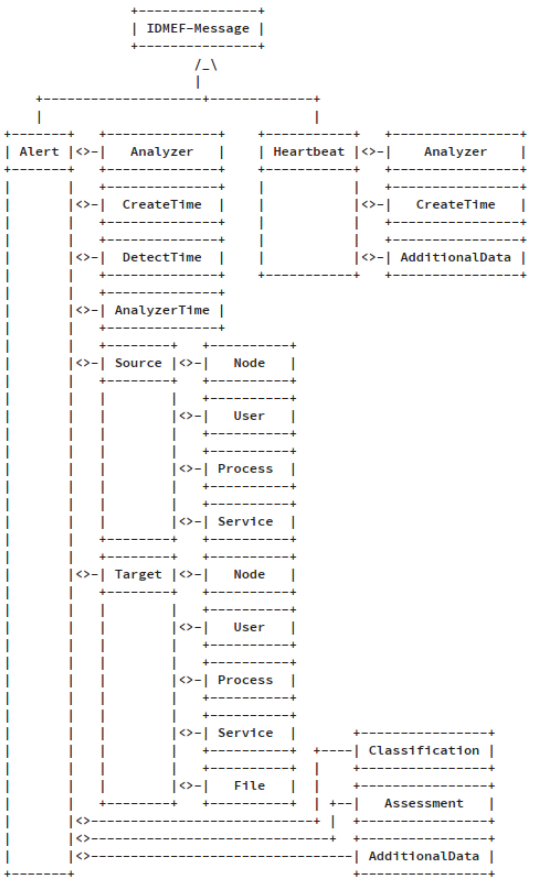

Figure A.1 – IDMEF data model, from [62].

Appendix B

Examples of IDMEF attacks representation

From [62], this is a network-based detection of a port scan. Shows detection by a single analyzer:

```
<?xml version="1.0" encoding="UTF-8"?>
  <idmef:IDMEF-Message version="1.0" xmlns:idmef="http://iana.org/idmef">
    <idmef:Alert messageid="abc123456789">
      <idmef:Analyzer analyzerid="hq-dmz-analyzer62">
    <idmef:Node category="dns">
      <idmef:location>Headquarters Web Server</idmef:location>
      <idmef:name>analyzer62.example.com</idmef:name>
    </idmef:Node>
      </idmef:Analyzer>
      <idmef:CreateTime ntpstamp="0xbc72b2b4.0x00000000">
    2000-03-09T15:31:00-08:00
      </idmef:CreateTime>
      <idmef:Source ident="abc01">
    <idmef:Node ident="abc01-01">
      <idmef:Address ident="abc01-02" category="ipv4-addr">
        <idmef:address>192.0.2.200</idmef:address>
      </idmef:Address>
    </idmef:Node>
      </idmef:Source>
      <idmef:Target ident="def01">
    <idmef:Node ident="def01-01" category="dns">
      <idmef:name>www.example.com</idmef:name>
      <idmef:Address ident="def01-02" category="ipv4-addr">
        <idmef:address>192.0.2.50</idmef:address>
      </idmef:Address>
    </idmef:Node>
    <idmef:Service ident="def01-03">
      <idmef:portlist>5-25,37,42,43,53,69-119,123-514</idmef:portlist>
```

```
    </idmef:Service>
      </idmef:Target>
      <idmef:Classification text="simple portscan">
    <idmef:Reference origin="vendor-specific">
      <idmef:name>portscan</idmef:name>
      <idmef:url>http://www.vendor.com/portscan</idmef:url>
    </idmef:Reference>
      </idmef:Classification>
    </idmef:Alert>
  </idmef:IDMEF-Message>
```

Same ports scan event alert as above but represented if it had been detected and sent from a correlation engine, instead of a single analyzer, example from [62]:

```
<?xml version="1.0" encoding="UTF-8"?>
  <idmef:IDMEF-Message version="1.0" xmlns:idmef="http://iana.org/idmef">
    <idmef:Alert messageid="abc123456789">
      <idmef:Analyzer analyzerid="bc-corr-01">
    <idmef:Node category="dns">
      <idmef:name>correlator01.example.com</idmef:name>
    </idmef:Node>
      </idmef:Analyzer>
      <idmef:CreateTime ntpstamp="0xbc72423b.0x00000000">2000-03-09
    T15:31:07Z
    </idmef:CreateTime>
      <idmef:Source ident="a1">
    <idmef:Node ident="a1-1">
      <idmef:Address ident="a1-2" category="ipv4-addr">
        <idmef:address>192.0.2.200</idmef:address>
      </idmef:Address>
    </idmef:Node>
      </idmef:Source>
      <idmef:Target ident="a2">
    <idmef:Node ident="a2-1" category="dns">
      <idmef:name>www.example.com</idmef:name>
      <idmef:Address ident="a2-2" category="ipv4-addr">
        <idmef:address>192.0.2.50</idmef:address>
      </idmef:Address>
    </idmef:Node>
    <idmef:Service ident="a2-3">
      <idmef:portlist>5-25,37,42,43,53,69-119,123-514</idmef:portlist>
    </idmef:Service>
      </idmef:Target>
      <idmef:Classification text="Portscan">
    <idmef:Reference origin="vendor-specific">
      <idmef:name>portscan</idmef:name>
      <idmef:url>http://www.vendor.com/portscan</idmef:url>
    </idmef:Reference>
      </idmef:Classification>
      <idmef:CorrelationAlert>
    <idmef:name>multiple ports in short time</idmef:name>
    <idmef:alertident>123456781</idmef:alertident>
    <idmef:alertident>123456782</idmef:alertident>
    <idmef:alertident>123456783</idmef:alertident>
```

```
    <idmef:alertident>123456784</idmef:alertident>
    <idmef:alertident>123456785</idmef:alertident>
    <idmef:alertident>123456786</idmef:alertident>
    <idmef:alertident analyzerid="a1b2c3d4">987654321
    </idmef:alertident>
    <idmef:alertident analyzerid="a1b2c3d4">987654322
    </idmef:alertident>
      </idmef:CorrelationAlert>
    </idmef:Alert>
  </idmef:IDMEF-Message>
```

Appendix C

Configurations

Esper configuration example, local correlator:

```
<?xml version="1.0" encoding="UTF-8"?>
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.espertech.com/schema/esper"
  xsi:schemaLocation="http://www.espertech.com/schema/esper http://www.espertech.
    com/schema/esper/esper-configuration-3.0.xsd">
  <event-type name="Addresses">
    <xml-dom root-element-name="//Address"
          default-namespace="http://iana.org/idmef"
        xpath-property-expr="false" resolve-properties-absolute="false">
      <namespace-prefix prefix="idmef"
        namespace="http://iana.org/idmef"/>
    </xml-dom>
  </event-type>
  <event-type name="IDMEF">
    <xml-dom root-element-name="IDMEF-Message"
      schema-resource="idmef-message.xsd"
      default-namespace="http://iana.org/idmef"
      xpath-property-expr="true"
      resolve-properties-absolute="true"
      auto-fragment="false">
      <namespace-prefix prefix="idmef" namespace="http://iana.org/idmef"/>
        <xpath-property property-name="MessageID"
          xpath="string(/idmef:IDMEF-Message/idmef:Alert/@messageid)" type="string
    "/>
        <xpath-property property-name="AnalyzerLocation"
          xpath="/idmef:Analyzer/idmef:Node/idmef:location/text()" type="string"/>
        <xpath-property property-name="AnalyzerName"
          xpath="/idmef:IDMEF-Message/idmef:Analyzer/idmef:Node/idmef:name/text()"
     type="string"/>
        <xpath-property property-name="CreateTimeText"
          xpath="/idmef:IDMEF-Message/idmef:CreateTime/text()" type="string"/>
        <xpath-property property-name="SourceAddresses"
          xpath="//idmef:Source/idmef:Node/idmef:Address" type="nodeset" cast="
    String[]" event-type-name="Addresses"/>
        <xpath-property property-name="TargetAddresses"
          xpath="//idmef:Target/idmef:Node/idmef:Address" type="nodeset" cast="
    String[]" event-type-name="Addresses"/>
```

```
        <xpath-property property-name="SourcePort"
          xpath="/idmef:IDMEF-Message/idmef:Source/idmef:Service/idmef:port" type=
    "nodeset"/>
        <xpath-property property-name="TargetPort"  xpath="/idmef:IDMEF-Message/
    idmef:Target/idmef:Service/idmef:port/text()" type="string"/>
        <xpath-property property-name="AdditionalDataString"        xpath="//
    idmef:AdditionalData/idmef:string/text()" type="string"/>
        <xpath-property property-name="ClassificationReferences"  xpath="/
    idmef:IDMEF-Message/idmef:Alert/idmef:Classification/idmef:Reference" type="
    nodeset"/>
    </xml-dom>
  </event-type>
  <auto-import import-name="java.lang.Double"/>
  <auto-import import-name="java.math.*"/>
  <!-- Bellow is to import the class defining the UpdateListeners annotation used
    in the correlation rules epl -->
  <auto-import import-name="eu.cockpitci.uc.correlators.server.esper.
    UpdateListeners"/>
  <engine-settings>
    <defaults>
      <logging>
        <execution-path enabled="true"/>
        <timer-debug enabled="false"/>
      </logging>
    </defaults>
  </engine-settings>
  <!-- Bellow is the class declaration of the AMQP EventBusAdaptor -->
  <plugin-loader name="EventBusInputAdaptor"
    class-name="eu.cockpitci.uc.correlators.server.eventbus.EventBusInputAdaptor">
  <!-- The parameter defined below declares the type of the correlator to be used
    by the EventBusInputAdaptor Global and Local correlator   can have different
    consumer connection parameters, correlatorType can have either "global" or "
    local" value -->
  <init-arg name="correlatorType" value="local"/>
  </plugin-loader>
</esper-configuration>
```

Correlator input adaptor configuration for global correlator:

```
BROKER_IP: 172.27.1.36
BROKER_PORT: 5672
VHOST: PIDS
USERNAME: guest
PASSWORD: guest
QUEUE_NAME: idmef.correlators
EXCHANGE_NAME: pids_exchange_global
EXCHANGE_TYPE: topic
EXCHANGE_DURABLE: true
EXCHANGE_AUTODELETE: false
ROUTINGKEY_DEFAULT: idmef.correlators
QUEUE_DURABLE: true
QUEUE_AUTODELETE: false
QUEUE_PASSIVE: false
QUEUE_EXCLUSIVE: false
ENABLE_DELIVERY_CONFIRMATIONS: true
RECONNECTION_ATTEMPTS: 3
RETRY_DELAY: 5
SOCKET_TIMEOUT: 500
SSL: true
```

Appendix D

Correlator statements

Event aggregation rule (event storm):

```
@Name("SourcesStream")
@Description("create a new event stream with event sources")
insert into SourcesStream
select srcaddresses, idmef.Alert.messageid as msgid,
  idmef.Alert.Analyzer.analyzerid as anaid,
  TargetAddresses as dstaddresses
from IDMEF[select address from SourceAddresses] as srcaddresses, IDMEF as
    idmef;

@Name("SegmentedBySourceAddressCtx")
@Description("create a context partitioned by source")
create context SegmentedBySourceAddress partition by srcaddresses.address
    from SourcesStream;

@Name("SourcesWindow")
@Description("create a named window for the context that keeps sources
    partition for the last 120 seconds")
context SegmentedBySourceAddress create window SourcesWindow.win:time(120
    seconds) as SourcesStream;

@Name("InsertSourcesWindow")
@Description("insert arrived events into the SourcesWindow")
context SegmentedBySourceAddress insert into SourcesWindow select * from
    SourcesStream;

@Name("EventStorm")
@Description("select sources with 100 events in the SourcesWindow, send
    results to IDMEFEventStormListener")
@UpdateListeners({"IDMEFEventStormListener"})
context SegmentedBySourceAddress
  select srcaddresses.address,
    context.key1 as evtstorm_source, msgid as id_message,
```

```
    anaid as id_analyser, dstaddresses as evtstorm_destinations
  from SourcesWindow
  output when count_insert = 100;
```

Listing D.1 – Event aggregation example (event storm detection)

Event filtering:

```
@Name("RateLimitSynFloodStatement")
@Description("Limits the output of SYN flooding statements from all Snort
    agents")
insert into AMQPOutgoingDataFlow
select Alert
from IDMEF(Alert.Classification.text='SYN Flooding')
where Alert.Analyzer.analyzerid regexp 'snort_agent-[0-9]+'
output first every 30 seconds;
```

Listing D.2 – Event filtering example statements

Event suppression:

```
@Description("Create variable to signal an alarm from OCSVM")
create variable boolean ocsvm_alarm = false;

@Description("Set variable ocsvm_alarm=true when receiving a SEVERE or
    MEDIUM alarm from OCSVM wihtin the last 5 minutes")
on IDMEF(Alert.Analyzer.analyzerid='oscvm-engine'
    and Alert.Classification.text in ('MEDIUM ALARM','SEVERE ALARM'))
    where timer:within(5 min)
  set ocsvm_alarm = true;

@Description("When an alrm is not received form OCSVM within the last 5
    minutes set ocsvm_alarm=false")
@Description("Create variable to signa alarm form OCSVM")
on pattern[every (timer:interval(5 min) and not IDMEF(Alert.Analyzer.
    analyzerid='oscvm-engine'
    and Alert.Classification.text in ('MEDIUM ALARM','SEVERE ALARM')))]
  set ocsvm_alarm = false;

@Name("HighArpAlert")
@Description("When receiving an ARP cache alert from snort and ocsvm_alarm=
    true forward event to output adapter")
insert into AMQPOutgoingDataFlow
select Alert
  from IDMEF(Alert.Classification.text = 'spp_arpspoof: ARP Cache Overwrite
     Attack'
    and ocsvm_alarm = true);
```

Listing D.3 – Event supression example statements