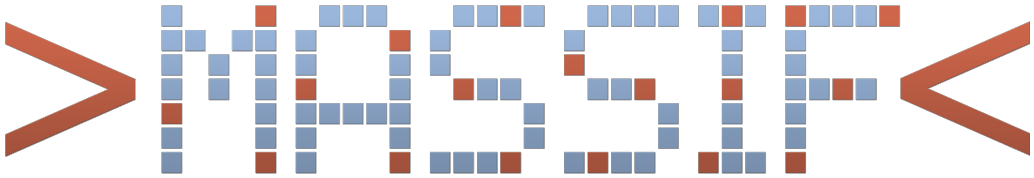

MAnagement of Security information and events in Service InFrastructures

MASSIF
FP7-257475

D5.1.4 - Resilient SIEM Framework Architecture, Services and Protocols

Activity	A5	Workpackage	WP5.1
Due Date	M36	Submission Date	2013-09-30
Main Author(s)	N. Neves (editor) (FFCUL), N. Kuntze (Fraunhofer) C. Di Sarno (CINI), V. Vianello (UPM)		
Contributor(s)	A. Bessani (FFCUL), R. Fonseca (FFCUL), M. Garcia (FFCUL) V. Gulisano (UPM), A. Mazzeo (CINI) , N. Mazzocca (CINI) R. Jiménez Peris (UPM), R. Rieke (Fraunhofer) , P. Rodrigues (FFCUL) L. Romano (CINI), P. Verissimo (FFCUL) , E. Vial (FFCUL)		
Version	v1.0	Status	Final
Dissemination Level	PU	Nature	R
Keywords	Resilient SIEM operation; Resilient communication and nodes; Resilient event storage; Attacks and accidental faults;		
Reviewers	Luigi Coppolino (Epsilon), Rodrigo Diaz (Atos)		

Part of the Seventh
Framework Programme
Funded by the EC - DG INFSO

Version history

Rev	Date	Author	Comments
V0.5	2013-07-01	Nuno Neves	First integrated version of the deliverable
V0.7	2013-07-19	Nuno Neves	Second integrated version of the deliverable, which includes improvements in all chapters
V0.8	2013-09-20	Nuno Neves	Text improvements due to comments by internal reviewers
V1.0	2012-09-30	Nuno Neves	Final review and official delivery

Glossary of Acronyms

AH	Authentication Header
AIK	Attestation Identity Key
API	Application Programming Interface
BFT	Byzantine Fault Tolerance
COTS	Component of the Shelf
CWE	Common Weakness Enumeration
DOS	Denial of Service
DoW	Description of Work
EC	European Commission
EG	Evidence Generator
ESP	Encapsulation Security Payload
EU	European Union
FEC	Forward Error Correction
FIFO	First-In First-Out
FP7	Seventh Framework Programme
FPR	Full Physical Replication
FVR	Full Virtual Replication
GPS	Global Position System
HTTP	Hypertext Transfer Protocol
ICT	Information and Communication Technology
ID	Identifier
IDS	Intrusion Detection System
IMA	Integrity Measurement Architecture
IP	Internet Protocol
IPsec	Internet Protocol Security
ISP	Internet Service Provider
IT	Intrusion Tolerant
LAN	Local Area Network
MAC	Message Authentication Code
MASSIF	MAnagement of Security information and events in Service InFrastructures
MIA	MASSIF Information Agent
MIS	MASSIF Information Switch
MTU	Maximum Transmission Unit
NVD	NIST National Vulnerability Database

OS	Operating System
OSI	Open Systems Interconnection
PCA	Privacy Certification Authority
PFR	Physical Filter Replication
PII	Personal Identifiable Information
PR	Proactive Recovery
PU	Public Usage
R&D	Research & Development
REB	Resilient Event Bus
RDF	Resource Description Framework
RSA	RSA algorithm for public-key cryptography
RTT	Round Trip Time
SIEM	Security Information and Event Management
SMR	State Machine Replication
SOA	Service-Oriented Architecture
SSL	Secure Socket Layer
TA	Time Authority
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TOM	Total Order Multicast
TPM	Trusted Platform Module
UDP	User Datagram Protocol
USB	Universal Serial Bus
VFR	Virtual Filter Replication
VM	Virtual Machine
VMM	Virtual Machine Manager
WAN	Wide Area Network

Executive Summary

Security Information and Event Management (SIEM) systems are being employed by organizations to facilitate operations related to maintenance, monitoring and analysis of networks and their nodes, by collecting and allowing the correlation of thousands of events. In particular, SIEM systems are used to report attacks and intrusions in near real-time, supporting recovery actions by the systems administrators. Since SIEM systems operate potentially over large geographically dispersed areas, where components (such as, sensors and collectors) are scattered through the existing infrastructure, they prone to various types of failures. Some of the inherent problems associated with this setting are delays, routing and node misconfigurations, event integrity violations and denial of service attacks, which can all affect the correctness of the event analysis.

This deliverable describes a framework for the resilient operation of a SIEM system, and then provides a set of specific solutions that can significantly improve the robustness of particular components. These solutions were developed based on the MASSIF SIEM needs, but in most cases they can be easily generalized to be employed in other systems. The document covers six areas: The architecture of a resilient SIEM; Robust event reporting addresses failures in the event collection, either by giving evidence that produced event data has not been tampered with or by addressing (some of) the existing problems while correlating the events; Resilient Event Bus (REB) enforces a correct communication among the edge-MIS and core-MIS devices under various failure scenarios; Node defense mechanisms explains techniques that can be applied in an incremental way to make nodes increasingly more resilient, and then uses them to build a highly resilient core-MIS; The fault tolerant Complex Event Processing (CEP) shows how to perform event correlation under accidental faults; Resilient Event Storage (RES) presents a solution for the secure archival of event data.

Contents

List of Figures

List of Tables

1 Introduction

Security Information and Event Management (SIEM) systems perform event collection by monitoring the network of an organization at various places, and then forward this information to a central location for correlation, supporting the discovery of attacks and malfunctions in near real-time. In some cases, remediation actions can be initiated immediately, allowing the automatic correction of the problems that were identified. Current SIEM systems carry out sophisticated forms analysis, including risk assessment and inventory management, and therefore, they are playing an increasingly more important role on the maintenance and administration activities of the organizations.

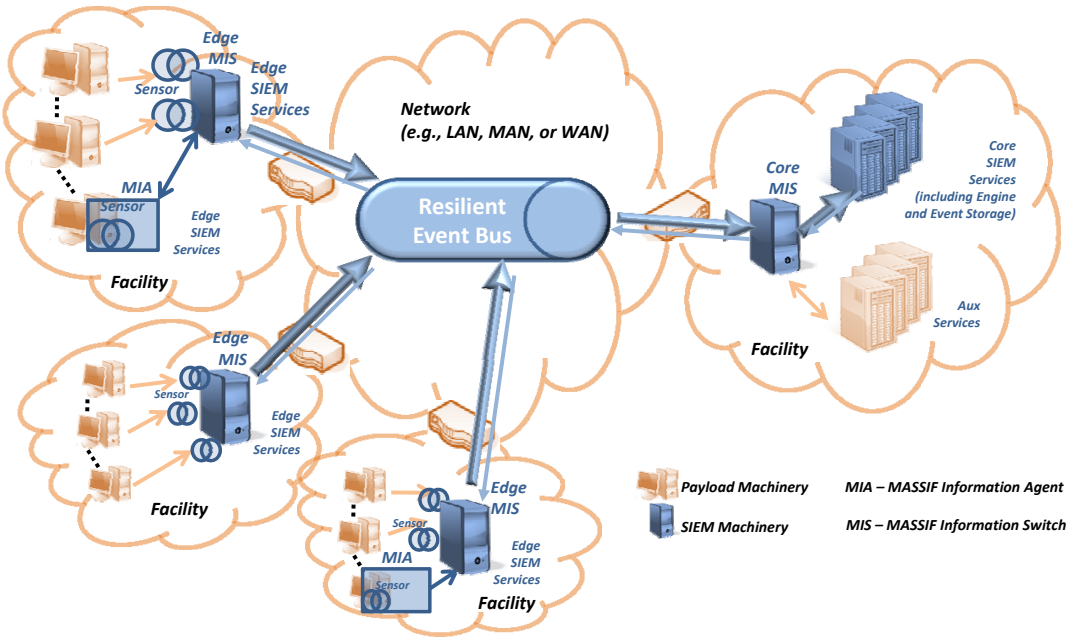

Figure 1.1: SIEM general architecture.

A SIEM system can be seen as an infrastructure superimposed on top of the existing network of an organization (called *payload machinery* in Figure 1.1). On the edges of the infrastructure are placed the sensors, which are responsible for generating the security events and for obtaining the relevant data about the monitored components (see left part of the figure). The monitored components can perform distinct tasks including network management and protection (e.g., routers, firewalls, intrusion detection systems), authentication functions (e.g., identity management systems) or support the execution of business related applications and services (e.g., web servers, databases). Events are locally forwarded to special nodes

called edge-MIS, which are placed in the vicinity of the sensors. At the edge-MIS, events can suffer some level of pre-processing, such as having their format normalized and/or being encrypted. Then, the edge-MISes cooperate to ensure that events are rapidly delivered to the core-MIS, which controls the traffic going in or out of the network where the SIEM main components are located (see the right side of the figure). The core-MIS next sends the data to the correlation engine for processing. Some of the events may also be archived in order to allow later analysis of security incidents or to support persecution actions against attackers.

Due to its fundamental role in the security management of an organization, the SIEM should be made secure and dependable when facing accidental faults and attacks. This deliverable consolidates the investigation carried out in the project, in order to devise an architecture and a set of mechanisms that can contribute to improve significantly the resilience of the MASSIF SIEM system operation. The document covers the following fundamental areas:

Resilient SIEM framework describes the guidelines followed in our design, whose basic idea is to ensure several levels of security and dependability; it also analyzes the main threat vectors to SIEM systems and propose an architecture and set of mechanisms to address them;

Robust Event Reporting covers two areas: it considers mechanisms that can be employed to give evidence that produced event data has not been tampered with, and therefore, that can effectively be used to make decisions with regard to the monitored systems; and, it provides an approach to construct more resilient correlation rules, in order to address various problems that might occur in event collection;

Resilient Event Bus (REB) enforces a resilient communication among the edge-MIS and core-MIS devices. The REB is based on an overlay superimposed on top of the existing SIEM infrastructure, which uses several mechanisms, such as coding algorithms, multihoming and multipath data transmission, to ensure data delivery in various failure scenarios;

Node defense mechanisms explains a set of techniques that can be applied in an incremental way to make nodes increasingly more resilient to different forms of faults, of either accidental nature or malicious attacks. Since the core-MIS plays a fundamental role in the protection of the core SIEM services, acting as a gatekeeper and preventing external attacks from entering in the network, we provide a design of how it could be built to be highly resilient by taking advantage of the proposed techniques;

Fault Tolerant Complex Event Processing describes how the correlation engine can detect and recover from crashes in the processing nodes, ensuring that no events are discarded or left unprocessed, and therefore, keeping the SIEM operational under failures;

Resilient Event Storage (RES) presents a solution for the secure archival of event data. RES is located together with the core SIEM services, and gives support for the forensic analysis of an incident based on the stored data, and enforces security policies that ensure that events are only made available to authorized entities according to the regulations.

1.1 Guidelines Analysis

According to requirements analysis and guidelines in deliverable D.2.1.1 [25], this deliverable contributes to the satisfaction of the following:

Guideline	Description
G.T.1. Resilience of the infrastructure	The various mechanisms presented contribute is different ways to this objective.
G.T.2. Security of event flows	The Authenticated Component Event Reporting, the REB, and the resilient design of the core-MIS all contribute to protect the event flows.
G.T.3. Protection of the nodes	Specific mechanisms are proposed for increasing the robustness of the nodes by introducing intrusion tolerance capabilities.
G.T.4. Timeliness of the infrastructure	The design of the REB and core-MIS have this guideline as one of their objectives.
G.T.5. Data authenticity	The various mechanisms presented contribute is different ways to this objective.
G.T.6. Fault and intrusion-tolerant stable storage	The RES design enforces this guideline.
G.T.7. Least persistence principle	The RES design enforces this guideline.
G.T.8. Privacy of forensic records	The RES design enforces this guideline.
G.S.6 Securing the evidence progressed by the MASSIF components	The Authenticated Component Event Reporting contributes to this guideline.

Table 1.1: Guidelines covered by this deliverable

1.2 Glossary Adopted in this Deliverable

As agreed by the MASSIF Consortium, the main reference of security glossary is provided by the National Institute of Standard and Technologies (NIST) [62].

1.3 Structure of the Document

The rest of the document is organized as follows: Chapter 2 explains the resilient architecture of the MASSIF SIEM. Chapter 3 presents the authenticated component event reporting solutions. Chapter 4 explains how to make correlation rules more robust to faults that might affect event collection. Chapter 5 describes the design and mechanisms employed by the REB to secure the communication among the MIS. Chapter 6 provides an explanation of techniques that can be used to improve the resilience of the nodes by making them intrusion-tolerant. Chapter 7 explains an approach to make event processing and correlation tolerant to node crashes. Chapter 8 presents the RES for secure event archival. The concluding remarks are provided in Chapter 9.

2 Resilient SIEM Architecture

Security Information and Event Management (SIEM) systems have gained significant importance in modern organizations, to a point that they became almost indispensable for managing the performance, dependability and security of ICT assets, especially in large installations. Some of the features that characterize these large infrastructures on which SIEM systems are often used are:

- highly distributed and large-scale, both in a geographical sense and with respect to the number of entities involved;
- heterogeneous, composed by end systems from possibly many vendors, with very diverse software and operating systems;
- heterogeneous networks interconnecting the end systems, with different levels of exposure, openness and trust.

However, current SIEM systems have some shortcomings in the way of trustworthiness, which stand in the way of the increased responsibility they are called for: event collection, dissemination and processing, is susceptible to attacks; centralized processing as a general rule creates bottlenecks and single points of failure.

From this opening scenario we extract three important observations:

- the monitored environments are rather complex and increasingly exposed to threats;
- the dependence on the monitoring systems increases, in the expectation that they ensure secure and dependable operation of the monitored systems, in real-time;
- the monitoring systems become themselves a potential target of attack, whilst being currently vulnerable and prone to different sorts of failures.

We are especially concerned with the last fact, which comes as a natural consequence of the first two. Given the shadow of advanced persistent threats hovering over critical installations, it is more than likely that the expected attack strategies include the "death of sentry", i.e., actions aimed at neutralizing the alert systems such as IDS or SIEM, before attacking the monitored systems themselves. In consequence, we advocate that *resilience*, understood as the capacity to maintain acceptable levels of security and dependability in harsh operating conditions, like persistent threats, possibly with incremental severity, should be considered a key attribute of SIEM systems.

The rest of this chapter is organized in the following way: We start by giving an overview of the architecture, itself intended to provide: (i) easy deployment of an SIEM system onto generic ICT systems; (ii) seamless integration of resilience into a distributed and large-scale SIEM system. We share with the reader the guidelines followed in our design, whose basic idea is to ensure several levels of

security and dependability in an open, modular, versatile and cost-effective way. We analyze the main threat vectors to SIEM systems and propose a set of mechanisms to address them. Finally, we elaborate on some example resilience mechanisms used in MASSIF, which are explained in more detail in later chapters.

2.1 Rationale for the MASSIF architecture

In this section, we establish the rationale for the MASSIF architecture through a list of basic objectives to be met by a generic and resilient SIEM system:

- Clear decoupling between the target (monitored) and SIEM (monitoring) system, for minimal impact on the observed infrastructure, and adaptation to varying target/SIEM system combinations;
- Enhancement of classical security techniques with resilience mechanisms against faults and attacks of incremental severity, maintaining availability, integrity and confidentiality;
- Promotion of automatic control of macroscopic information flows;
- Avoidance of single points-of-failure;
- Reconciliation of resilience with legacy preservation;
- Preservation of timeliness (real-time operation) in the presence of faults and attacks;

2.1.1 Objectives

We briefly elaborate on those objectives. The first objective imposes a generic architectural goal for modern SIEM systems, which are bound to encompass multiple ICT infrastructures, achieving a global span. An effective solution should be as independent as possible from the observed system, both to disturb the target the least possible, and to preserve the SIEM system from direct threats.

The next step is based on the observation that classical security techniques are largely based on prevention, human intervention and ultimately disconnection. Given the complexity of systems and the speed at which attacks develop, there is a growing need for achieving instead, *tolerance, automation and availability*, both under attack and in the presence of major accidents.

The next three objectives further refine the above-mentioned propositions. The workhorse of resilience are fault and intrusion tolerance mechanisms, essentially based on redundancy, to prevent the collapse of the system on account of a single point of failure, and the use of automatic mechanisms to effect error processing. With such techniques, securing the correctness of the command and information flows between major modules becomes straightforward, through so-called intrusion-tolerant protocols. Redundancy and diversity both purposely introduced and derived from the sheer infrastructure richness and complexity, should be used to devise these fault and intrusion tolerance mechanisms, keeping the system working despite the failure or malicious compromise of individual components. Likewise, resilience solutions should, in turn and as much as possible, be transparent to the functionality of the SIEM system and, in consequence, to the payload system, preserving legacy.

Last but not least, existing systems have a hard time reconciling security with timeliness. Security solutions in distributed systems tend to be asynchronous, that is, not relying on timeouts or other timing

bounds, in order to reduce the attack plane given to adversaries. However, an SIEM system, providing a real-time view and requiring real-time capability of reaction, cannot afford to trade-off timeliness for security. This is easier said than done and in fact, attacks on the timeliness properties of the information flows coming from the collection points in the edge to the processing engines in the core (e.g., selective delays and reordering) may neutralize the correlation capability of the event processing engines. This hard problem is also tackled in MASSIF.

2.1.2 Main characteristics

We propose to address these requirements by an architecture which we describe below, having the following main characteristics:

- A topology following the WAN-of-LANs model [104], and laid down as a logical overlay over the target or payload system, so as to preserve legacy but allow seamless integration of the monitoring and monitored systems, possibly across different and wide-scale administrative domains.
- Modular and adaptive structure, achieved by: (i) using modular functions and protocols, to be re-used by different instantiations of the architecture; (ii) perform a good separation between monitoring and monitored systems, by concentrating these functions in configurable conceptual devices which act as the nodes of the overlay: MASSIF Information Switches (MIS).
- Information flow in the overlay implemented as a secure and real-time event bus, modeled essentially as a producer-consumer SCADA-like system upstream, with low-bandwidth commands downstream.
- Resilience procurement based on: securing the information flow; making the dissemination infrastructure itself (event layer) resilient; protecting crucial processing units (MIS) with incremental resilience strategies relying on hardware and software based alternatives; and differentiating between edge-side and core-side configurations.

2.1.3 Structural view

As proposed above, the MASSIF architecture offers a topology relating the payload system (monitored system) to the SIEM system (monitoring system), through an infrastructural overlay on the former. The topology of a MASSIF SIEM system is shown in Figure 1.1. We can observe the MASSIF SIEM system (blue) as an infrastructural overlay of the monitored system (brown). The overlay is implemented by the MASSIF Information Switches (MIS), which provide the dual functionality of being the hooks to the monitored system and serving as communication nodes of the overlay.

The hooks or contact points between both are clearly materialised by the MIS, serving standard protocols to interface with the payload system. The MIS on the edge-side implement classical *collector* functions, getting events from the monitored system. The core-side MIS implement protection functions like *firewalls*, securing the core services perimeter. All the MIS together support the overlay communications fabric implementing the resilient event bus, through MASSIF's own fault and intrusion tolerant protocols, which offer timeliness and resilience characteristics as discussed earlier.

Note that the payload system can retain its essential characteristics when the SIEM infrastructure is superimposed on it, since both work essentially in parallel. As represented in the figure, some collector

functions can be delegated onto payload system devices, to software agents, MASSIF Information Agents (MIA), implementing *smart sensors*.

A WAN-of-LANS model, as depicted in the figure, is a useful construct to represent the reality of the large ICT infrastructures we see served by SIEMS, typically made-up of several asymmetric and loosely-coupled facilities, sometimes widely separated geographically, whose local intranets are interconnected through public networks like the Internet, possibly under the protection of secure channels or tunnels. By asymmetry we mean varying degrees of security, dependability or performance. Furthermore, it becomes easy to decouple the threat scenarios faced by the WAN part from the LAN parts and, moreover, it is quite simple to consider different levels of trustworthiness for different selected facilities, as well as individual LANS inside those facilities. The 'LAN' concept is just a comfortable abstraction with the adequate granularity to mean "short-range", whose real expression normally involves switching or routing topologies at layers 3-1.

2.2 MASSIF system components

The main building blocks of the architecture are now explained with more detail (Figure 2.1).

Data layer. The aim of the services in the Data layer is to collect the relevant security data from the payload layer machinery (i.e. the lower layers devices supplying raw security information and event data). The Data layer provides services for the collection of sensory information from the monitored environment, for the processing and filtering out of non-relevant information, and for the final normalization of the events.

MASSIF events with different formats and origins and from different application domains, need to undergo a process of abstraction and coalesce into what we call MASSIF Generic Events, following a common syntactic and semantic format, which allows events from anywhere in the infrastructure to be fused into the Event Bus, which performs reliable and ordered event dissemination, and gives services at the Application layer a convenient way to treat this uniform event flow.

Another key service provided by the data layer of the MASSIF architecture is pre-correlation at the edge: the objective of pre-correlation is to transfer part of the SIEM intelligence to the edges of the architecture in order to balance the load on the core processing engines and to reduce the communication traffic. Last but not least, reaction and adaptation services can be provided by agents at the egde, responding to downstream commands from core SIEM services such as decision support and reaction application modules. They may serve, for example, for configuration and reconfiguration of the sensing policies.

Event layer. The Event layer services essentially support the reliable flow of information and also control communication between MASSIF nodes, guaranteeing that this service is resilient both to accidental and malicious faults. They can also implement protection, for example of core services attached to extremities of the Event layer nodes (MIS). When needed, MIS are dual-homed, implementing, besides the SIEM functionality, a resident protection service akin to an application-level firewall, acting as a bastion providing perimeter defence. As mentioned earlier, this is especially interesting for core-side specific critical subsystems, such as the core SIEM event processing engines.

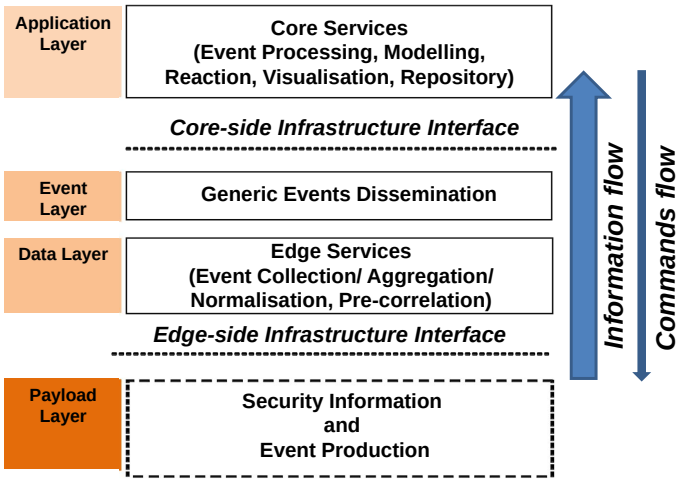

Figure 2.1: MASSIF architecture block diagram.

Application layer. The services offered by the Application layer are the MASSIF core intelligence: they essentially implement efficient and effective ways to derive information from the massive flows of event data arriving from the edge. SIEM classical application services are manifold. Event Processing is a main service, implemented by a highly scalable event processing engine, being responsible of processing large amounts of streaming data in real time, as well as stored events for forensic analysis, with multi-level abstraction and correlation capabilities based on user-defined rules in a distributed, efficient, elastic and scalable way. Modelling and Simulation services implement new process/attack analysis and simulation techniques in order to be able to dynamically relate events from different execution levels, and interpret them during runtime, in context of specific security properties. Two main approaches followed in MASSIF are Predictive Security Analysis and Attack Modeling and Security Evaluation. Decision Support and Reaction services allow to implement consolidated security policies across the different components, in a dynamic way, automatically configuring those components based on selected countermeasures.

The bulk of MASSIF distributed services described reside between what we call Edge-side Infrastructure Interface and the Core-side Infrastructure Interface (Figure 2.1), and are implemented by the MIS. The payload layer offers raw security information and event data, at the Edge-side Infrastructure Interface. This information is collected by the MIS (or MIA). Individual MIS (or MIA) implement services related to the Data layer, such as event collection, aggregation and normalisation. The collection of MIS devices then run the secure, reliable and real-time communication protocols needed to implement the Resilient Event Bus (REB), performing generic events dissemination to the core services, available at the Core-side Infrastructure Interface. The core SIEM engines at the Application layer run the services described above, like correlation, modelling and simulation, decision support and reaction, as well as ancillary services like storage and GUI.

2.2.1 Edge-side Services

In the MASSIF model, we consider that edge-side monitored system payload devices actually have their own basic sensory apparatus in place, be them raw event sources/emitters — e.g. logs — or native sensors — purposely made metrology artefacts that measure alarm conditions of the payload systems. These are normally supplied with the monitored systems, even if they are extensions to basic configurations. We will generally call (monitored system) sensors to whatever is in place to acquire the raw security information and event data from the payload.

The edge-MIS then act as collectors of information from the payload sensing apparatus, at the Edge-side Infrastructure Interface[1]. Each edge-MIS is then in charge of implementing part or all of the Data services foreseen in MASSIF, which perform some sophisticated data processing. Namely, an edge-MIS should do at least event collection. However, it will normally implement other services as well, namely the normalization of the event formats and contents, aggregation of several events, and even some local pre-processing and correlation. Likewise it may also host agents capable of performing reaction and adaptation commands. Generally speaking, we talk of MASSIF smart sensors, to address the edge-MIS data modules that acquire and process the basic information coming from payload sensors. This duality is shown in Figure 1.1.

Additionally, MIS being typically implemented as stand-alone machine/devices, for modularity, ease of configuration, performance and protection reasons (we may think of a MIS as an appliance box plugged onto the network), we also foresee software implemented versions of the same module, which we call MASSIF Information Agents (MIA). An MIA is a software appliance residing in edge payload nodes. The essential difference between the edge-MIS and the MIA depicted in Figure 1.1, is that the first is implemented by a "box" which resides on the network and can be addressed by any device of the payload, through standard protocols like TCP/IP. This is the standard situation, where MASSIF SIEM relies on the payload's own sensors, and does not involve any modification of the information-producing devices, which send their log, event or alarm files to the nearest edge-MIS.

On the other hand, the MIA implements a remote smart sensor, that is, a MASSIF compliant sensor which allows part of the data layer functions to be performed in the payload machinery. This requires payload nodes to offer a local API to the basic sensing apparatus (syslogs, event services, etc.), and be open to installing external software modules, but apart from that, it should require minimal host modifications, allowing swift integration of MASSIF functionality into non-closed payload nodes.

The additional integration effort of MIA into selected existing payload devices may well be justified for nodes offering reasons for local MASSIF intelligence: critical nodes such as core routers; nodes that are themselves very rich in information and event sources. As a matter of fact, certain devices, such as firewalls or IDS (Intrusion Detection Systems) are so rich and sophisticated in the information they provide, that it makes sense to incur the cost of porting (some of) the MIS services to a software module compliant with the architecture of the former. Another reason for resorting to an MIA is when a given payload device, albeit important, does not have incorporated sensors (i.e., lacks software modules capable of generating syslogs, events, etc. in a format exportable or understandable to the MIS). This will be rare in ICT, but may happen in control devices such as used in critical infrastructures. A slightly higher integration effort may be well justified for critical devices lacking sensing capability.

In any case, one of the advantages is the capability of pre-processing and filtering the information, and even tuning those firewall or IDS devices in special ways, in response to commands issued by

[1]For the sake of taking advantage of the architecture asymmetry, we separate between edge and core MIS which, though similar in nature, may have different configurations, be treated differently e.g., by producer-consumer protocols, and/or have different complexity and resilience.

the Application layer. Another advantage of the MIA approach is guaranteeing a more trustworthy information and event feed from/to that particular payload node, not subject to the communication faults and attacks discussed in Section 2.3, since MIA-MIS interconnection is made through MASSIF reliable communication protocols.

2.2.2 Resilient Event Bus

MASSIF Information Switches also play an important role as generic communication servers, namely implementing the Resilient Event Bus, REB. The collection of MIS devices run the secure, reliable communication protocols needed to implement the Resilient Event Bus abstraction. These protocols can use essentially the same kind of substrate of communication as the payload system. More secluded architectures for highly critical applications can nevertheless be foreseen, with dedicated secure circuits or virtual private networks to implement the REB. Though this component will henceforth be designated Resilient Event Bus, the resilience aspects will be discussed later in Chapter 5, whereas here we introduce the functional aspects.

The Resilient Event Bus (REB) is mainly in charge of disseminating the events collected by the edge-MIS, after being pre-processed by the Data services implemented in the same edge-MIS, to the core-side Application layer services. The trustworthy MIS-to-MIS interconnection secures these information flows. The REB delivers the information to the core-MIS, which communicate reliably with the core engines, at the same time protecting them from external attacks, acting pretty much as a sophisticated firewall.

The REB should encompass both events created by the periphery and events generated from within the SIEM machinery. As shown in Figure 1.1, events are put into the event bus, mainly by the edge-MIS, to be delivered to the core-MIS subscribers. But this does not preclude edge-MIS from subscribing, or core-MIS from sending events. In fact, that happens each time there are notifications or commands sent from the core services down to the edge of the infrastructure. The flow from edge to core, as the figure suggests, is expected to have much greater bandwidth than the flow in the opposite direction, used to carry commands in reaction to the analysis performed by the correlation engines and other application services. In fact, given the latency and throughput demands of the expected event flows from the edge to the core, the event will push the information in near real-time from the sender to the core entities having subscribed to it, the Event Processing Engine being the main subscriber.

2.2.3 Core-side Services

Core SIEM services are for example the event processing engine, and other services like modelling, decision support and reaction, visualisation, repository. The core application services process the information and events arriving from the edge, performing complex event analysis, normally using the stream data processing model, trying to find correlations in the data and detect anomalies (failures, intrusions). Besides correlation, data is also archived in resilient storage, in order to allow ulterior forensic analysis. Reaction modules may generate commands to modify the sensing and collecting conditions, or even modify protection or filtering apparatus like firewalls or IDS, namely those mediated by MIA. Auxiliary services are any non-critical services that are not part of MASSIF, but may be of interest to the operation of MASSIF as a whole (long-term archival, email, web apps, reporting, printing, etc.).The application services are bound to reside in data centers either of the monitored system's organisation (running its

own MASSIF SIEM system) or of a third party organisation (in the case of an outsourced MASSIF SIEM managed service). Remember that MASSIF is supposed to operate through diverse administrative domains, and this is one of the reasons.

As Figure 1.1 suggests, these core-side critical subsystems (Core SIEM services), for example core SIEM event processing engines, are supposed to be housed in perimeter-protected LANs connected to the MASSIF WAN-of-LANs. As mentioned earlier, dual-homed MIS can implement this protection: as depicted in the figure, such core services lie behind a MIS, which filters all access, both from the network and from the facility intranet. In fact, with regard to the latter, note that the Auxiliary services, which belong to the payload, can only interact with the core services via a MIS.

2.3 Why resilience? Understanding attacks and faults on SIEMs

This section deals with the observation made previously that, as dependence on the monitoring (SIEM) systems increases, they become themselves a potential target of attack, and our more than probable prediction that sophisticated attackers will aim at neutralizing the alert systems such as IDS or SIEM, before attacking the monitored systems themselves.

This situation being unavoidable, our point is that the problem is aggravated by the fact that current SIEMs are generically vulnerable and prone to different sorts of failures. Since SIEM subsystems have today a highly distributed nature and operate in essentially the same environments as the monitored systems, they can easily fall prey to attacks and even serious accidental faults. In order to prove our point, we analyse the susceptibility of an SIEM system, being a distributed and large-scale architecture with a high level of exposure, to faults and attacks, some of which of possibly large and/or uncertain magnitude.

We specifically determine what are the potential attack vectors to an SIEM, as depicted in Figure 2.2:

1. sensing flow integrity, which typically uses standard protocols (arrow 1) — e.g., tampering with the standard protocols conveying information (e.g., SYSLOG) from devices to the collector: interrupting, delaying, re-ordering, replaying, forging, etc.;

2. collector, targeting its availability and/or the integrity of event collection and/or communications (arrow 2) — e.g., disruption (DoS) or penetration attacks on collector: SIEM services and/or communication protocols;

3. agents, with the objective of attacking their availability and/or the integrity of remote event collection and/or agent-to-colletor communications (arrow 3) — e.g., disruption (DoS) or penetration attacks on device-resident agent: SIEM services and/or communication protocols;

4. Event Bus, targeting its confidentiality, integrity and availability (arrow 4) — e.g., tampering with the protocols conveying information between collectors and core systems: interrupting, delaying, re-ordering, replaying, forging, etc.;

5. firewall, aiming at attacking its availability and/or the integrity of the protection service and/or the communications (arrow 5) — e.g., disruption (DoS) or penetration attacks on firewall: SIEM services and/or communication protocols;

6. core systems, targeting their availability and/or integrity (arrow 6) — e.g., disruption (DoS) or penetration attacks on core services (SIEM Engine, Historian, GUI, etc.);

7. auxiliary services, targeting integrity of interactions (arrow 7) — e.g., disruption (DoS) or penetration attacks on auxiliary services.

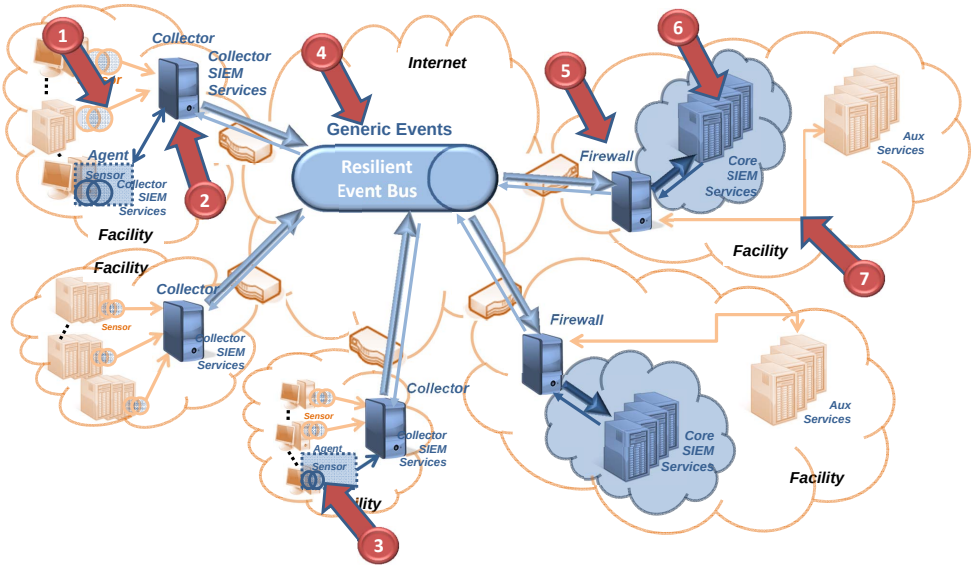

Figure 2.2: Attack vectors to an SIEM architecture.

We conclude that there is ample margin of attack to current large-scale and open SIEM systems. These attack vectors have to be prevented from compromising the correctness of the SIEM system. Therefore, it is important to improve the trustworthiness of SIEM systems, by employing the appropriate resilience mechanisms and protocols to safeguard the operation of the nodes and the communications. The discussion below applies to the MASSIF architecture, as our proposal for a reference modern SIEM architecture. However, several of the proposed mechanisms are susceptible of being integrated in other kinds of SIEM architectures.

2.4 Resilience mechanisms for SIEM systems

How to secure an SIEM architecture? A systematic approach will look at the information flow and secure its key parts, mitigating the consequences of the attack vectors identified in Section 2.3: (i) threats to the communication path; (ii) threats to the communication elements themselves; (ii) threats to the processing elements. The intent of this section is to describe the main techniques that are being explored in MASSIF to improve its resilience.

In our approach to increase resilience, we will employ prevention techniques whenever possible to deal with various types of threats, such as eavesdropping and/or tampering of messages. For instance, traditional cryptographic solutions based on in symmetric encryption and Message Authentication Codes (MAC) are highly effective at averting this sort of attacks, and nowadays they provide efficiency levels that can address information flows with huge amounts of events.

However, some more severe attacks are hard to solve with prevention solutions alone (e.g., an intrusion in the core-MIS machine), and therefore, it advisable to employ mechanisms to achieve tolerance [105, 106]. In short, instead of trying to prevent every single intrusion or fault, they are allowed, but

tolerated: systems remain to some extent faulty and/or vulnerable, attacks on components can happen and some will be successful, but the system has the means to trigger automatic mechanisms that prevent faults or intrusions from generating a system failure.

Additionally, while disconnection can be an effective solution to avoid the propagation of attacks, it may imply significant performance degradation and may have very negative and costly implications to service provision. It is thus important to seek for solutions that allow availability under attack.

Given that, as we have assumed earlier, different facilities/networks of the payload and the SIEM system may have asymmetric levels of trustworthiness, and that distinct application and systems will require different levels of trust, the architecture must allow for an incremental range of resilience solutions, in the interest of the best trade-off with performance, cost, or complexity.

For example, in MASSIF we discuss techniques to improve the resilience of specific nodes of the architecture, such as the aforementioned MASSIF Information Switches (MIS). The MIS can be built with incremental levels of resilience, depending on its criticality, from baseline OS-hardened simplex machines [94], up to physically replicated Byzantine-fault resilient units [16]. Recall that we leave the monitored system essentially untouched, and base our resilience solutions on the overlay, of which the MIS are key points, implementing collectors, firewalls, and communication servers.

The communication among the MIS plays a fundamental role in the MASSIF resilience architecture. This feature is responsible for delivering events from the edge services to the core SIEM correlation engine despite the threats affecting the underlying communication network. The Resilient Event Bus is an overlay communication subsystem internal to the MASSIF SIEM and thus itself protected, much in the sense that secure VPN (virtual private networks) are. To give this kind of guarantee we will employ application-level routing strategies among the MIS nodes, in such a way that they form an overlay network able to deliver messages in a secure and timely way.

The MASSIF architecture allows for multiple strategies for protection of the core components executing application layer services. The simplest one is perimeter defence, by isolating the core components within trusted intranets, only communicating with the outside through a MIS, in two ways: with the Resilient Event Bus; and with auxiliary systems. Besides executing protection functions, the core-MIS is itself built with resilience enhancing mechanisms, to protect it from direct attacks. Besides this baseline protection, SIEM core resilience can be enhanced through more sophisticated forms of protection, through fault and intrusion tolerance. Such solutions would for example provide resilience against insider attacks. The nature of the Resilient Event Bus communication model extends the modularity of the edge subsystems to the core systems: application servers may actually reside in more than one protected intranet, offering a multitude of deployment and server placement strategies.

The SIEM event processing engine deserve special attention, since it is the most data intensive of all core-side, application services. As the engine may need to support high loads of events, the mechanism to improve resilience needs to provide low runtime overhead and fast recovery. Runtime overhead must be kept as lower as possible as the mechanism is useful as long as its impact on the normal processing does not violate the application requirements. On the other hand, recovery time should be as short as possible in order to reduce the quality loss and the user satisfaction upon failures. Consequently, this imposes strict requirements on the mechanisms that are selected, precluding some of the approaches used in other components, for instance, the ones based on intrusion tolerance replication. In fact, one can leverage from the protection provided by the core-MIS, to resort to mechanisms that focus on tolerating accidental faults.

The storage solutions to be deployed in the MASSIF architecture have several purposes, requiring different levels of resilience. Amongst them, MASSIF foresees storage units dedicated to archival of critical security information and events, requiring properties like integrity, confidentiality and unforge-

ability. One of the obvious uses of such resilient storage is to archive important security information and events in a way justifiably usable for criminal/civil prosecution of attackers after a security breach.

3 Authenticated Component Event Reporting

Various equipment, including the sources of event data, relevant for the operation of the overall infrastructure is placed in non-protected environments. This recent development can be observed for example in smart grids for energy distribution or approaches in the area of facility management. It is therefore possible for attackers to acquire access to equipment with relative ease, and then initiate fake event reporting. This chapter studies the impacts of this problem and suggests solutions to address it.

In particular, this chapter describes the following two results from the work package WP5.1:

- Methods for authenticated component status reporting
- Mechanisms for unforgeability provision to support criminal/civil prosecution

Based on these considerations, this chapter furthermore briefly describes a prototypical implementation of a trusted information agent which has been done by the MASSIF user group at the Technische Hochschule Mittelhessen (THM). This implementation is not part of the core MASSIF architecture but can be considered as an optional component to improve the non-repudiation and reliability in events from external sources placed in non-protected environments.

3.1 Problem description

We will start by analyzing some possible misuse cases, which have been reported in the scenario deliverable [25] of the MASSIF project.

Water level sensor compromise The attacker takes control of the water level sensors and uses them to send spoofed measurements to the dam control station. This hides the real status of the reservoir to the dam administrator. In this way, the dam can be overflown without alarms being raised by the monitoring system.

From this, we get the requirement that the water level measures have to be authentic for the administrator when they are displayed at the dam control station.

Tiltmeter compromise The attacker takes control of the tiltmeter sensors and uses them to send false measurements to the dam control station, thus hiding the real status of the tilt of the dam's walls to the dam administrator. An excessive tilt may lead to the wall's failure.

Crackmeter / jointmeter compromise The attacker has access to one of the crackmeters or jointmeters deployed across the dam's walls and takes control of it. So the attacker can weaken the joint or

increase the size of the crack at the wall's weak point without any alarm being raised at the monitoring station.

These examples show the need for respective authenticity requirements. In general, however, information flows between systems and components are highly complex, especially when organisational processes need to be considered. Hence, not all security problems are discoverable easily. In order to achieve the desired security goals, security requirements need to be derived systematically [34].

In summary, the analysis of the use case and misuse cases of this critical infrastructure scenario shows that the overall function of the system requires authenticity of measurement values for several sensors. In that sense, the dam scenario is a prime example for the relevance of devices which satisfy respective authenticity requirements.

Authenticity requirements and devices A data record can be considered secure if it was created authentically by a device for which the following holds:

- The device is physically protected to ensure at least tamper-evidence. The data record is securely bound to the identity and status of the device (including running software and configuration) and to all other relevant parameters (such as time, temperature, location, users involved, etc.[3])
- The data record has not been changed after creation.

Digital Evidence according to this definition comprises the measured value (e.g., water level sensor measurement) and additional information on the state of the measurement device. This additional information on the state of the measurement device aims to document the operation environment providing evidence that can help lay the foundation for admissibility. As in the case of calibration of breathalyzers, for example, if the measurement device is modified, such information should also be recorded as part of amassing information supportive of admissibility. This will permit, at a later date, the linking of the software version used to collect the evidence in question. This information would permit an expert witness to testify to the known vulnerabilities of that particular software version and thus the likelihood of attacks.

Forensically Ready. By incorporating requirements into device design that focus on 1) potential admissibility of data records created by the device and 2) creating additional documentation that would support arguments for admissibility, we establish devices that are 'forensically ready'. Subsequent transport and secure storage of digital evidence are not part of this discussion, although they must be considered by anyone responsible for operating a network in a manner that ensures collection of competent legal evidence. However, for the purposes of this deliverable, we assume that digital evidence is created and stored in the device in question, and that there exists reliable mechanisms to maintain authenticity and integrity of the data records and also to provide non-repudiation for any steps of handling or changing the data, perhaps relying on some kind of digital signature which is often the case. For long-term security, archiving schemes can be used where digital signatures are replaced with some other security mitigations, anticipating that employed cryptographic algorithms will become unreliable due to increasingly sophisticated attacks or evolving computing capabilities. Physical attacks on devices are also not included in the discussion. We are assuming that, as in many cases, it will be sufficient to install tamper-evident devices (e.g., by using sealed boxes, installing devices in physically controlled rooms, etc.). Constructing real tamper-proof devices is expensive and difficult. Thus we are focusing on security at the mechanism level– how we develop and implement requirements for forensic readiness. Digital evidence requires

[3]The actual set of parameters and the protection levels depend on the scenarios and on the type of data record

that additional security mechanisms be implemented into the hardware that will render them impossible to be manipulated without physical access to the device.

The aim of this task is to integrate industry approaches to the attestation of event reporter states and how to integrate these measurements to gain a certain degree of trustworthiness and non-repudiation for events collected.

3.2 Technical solutions for the creation of digital evidence

The legal requirements towards the creation of digital evidence as discussed above imposes strong requirements on the security of individual technical devices as the evidentiary records but also on the processes for validating the device and software running on the device, for transmitting and storing evidence records, for linking evidence records to a chain of evidence, and also for verifying evidence records in the case of a dispute. The following subsections provide an overview of existing technical approaches starting from securing the actual creation on the individual device, looking at the infrastructure, and finally the processes involved.

3.2.1 Individual device

Devices with various interfaces pose particular problems. Besides typical communication network interfaces, direct or close-range access via USB, for example, increase the complexity of protecting devices from physical access, let alone network attacks. As discussed previously, the complexity of current state-of-the-art devices presents a challenge for constructing a secure device that is both efficient and useable. Therefore, taking a pragmatic approach to securing digital evidence on these devices, we suggest to focus on establishing assurance that the device was not manipulated at the time of the creation of the evidentiary record.

One approach might be to establish a cryptographic binding of evidence to the status of the device [90]. This can be achieved by using the existing technology of Trusted Computing [78] as specified by the Trusted Computing Group. The so-called Trusted Platform Module (TPM) can establish a hardware root of trust in the device. The security-level of a specialized security chip can be compared to SmartCard security. In combination with a first trusted step in the boot process, the TPM can be used to store, and securely report, measurement values documenting all software that was loaded after the current boot started. Further, the TPM provides the functionality to sign data records combined with these measured values and also to time-stamp data records to reliably reflect time relationships. The first prototypes of traffic cameras secured by this technology are available [110, 109]. In addition to the so-called attestation of the current boot process of a device as established by Trusted Computing, there also exist approaches that go beyond attesting to only one boot cycle. The cumulative attestation proposed by LeMay and Gunter [68] provides additional records and attests to the history of the boot process.

In contrast to the Trusted Computing approach, measurement values are not completely deleted for each re-boot, but a cumulative measurement chain is generated over several boot processes. This approach ensures that the device has not been booted in an insecure start after the cumulative measurement has started. It should be noted that, by using hardware-based roots of trust protection, this also prevents some types of insider attacks where insiders try to produce false evidence. The trust in the status reporting of a particular device is rooted in certain core roots of trust. The TPM is one prominent example of an available root of trust for reporting. These roots of trusts are built and certified by public bodies to

be tamper-proof, or at least hard to tamper. This reduces the possible attack vectors that could result in modification of the reported status of the device, even to authorized insiders like administrators.

3.2.2 Infrastructure

It should be noted that securely creating a data record is not sufficient to establish secure digital evidence. The device producing the record must be integrated into an appropriate infrastructure that can be structured into two parts: 1) elements that collect the data that then is stored in the evidence record and 2) securely transmitting and maintaining long-term storage of that data. Data collection is not only about maintaining the integrity of the data. Correctness of sensor data depends on many other factors, such as physical parameters of the environment (e.g., temperature or humidity), the location of the device and the physical integrity of the sensor itself. Some of these factors can be controlled by additional sensors; the status of these could be included in the reporting from the hardware-based attestation mechanisms.

Nevertheless, physical manipulation of the sensors is always possible. Threat modeling and risk analysis can provide analysis of residual risks remaining after Trusted Computing is implemented. Integrity and authenticity of data records can be maintained through use of public key cryptography. Since the private key can be stored exclusively inside a hardware security chip, this aspect of the infrastructure can be secured in this manner. Also solutions for long-term archiving exist (e.g., by renewing digital signatures before their algorithms are broken and signatures become useless). The mechanisms for this type of protection are well-established and can be efficiently implemented. However, digital evidence can contain Personal Identifiable Information (PII), requiring application of privacy enhancing technologies to digital evidence. Additional infrastructure is needed if several individual evidentiary records are linked to a *chain of evidence* [64].

3.2.3 Process

In addition to technical solutions for securely creating and storing digital evidence and digital evidence chains, organizational processes must enable the correct implementation and reproducibility of these technical solutions. Verification and checking of digital evidence cannot be restricted to checking a single digital signature per evidence record. It also needs to include additional checks on cryptographic key certificates and validation of the status of the devices involved in the creation of evidence records. Various types of digital certificates for cryptographic keys or software measurement values will be necessary. Additional checks can be required such as certification of the platforms involved in the creation of evidence records. A chain of evidence (or most probably a tree or several linked trees) would require going through this process for each type of digital evidence and to establish all necessary links between evidence records.

The following describes a proposed procedure required in advance of actually producing signed digital evidence:

1. Produce hardware security anchor (e.g., TPM): The hardware security anchor must be produced at a high security level.

2. Certify hardware security anchor: Security properties of the hardware security anchor should be documented in a security certificate with an appropriate security level.

3. Certify platform: In addition to the single security chip, the means of its integration into the platform and the properties of the root of trust for measurement are relevant and should be verified and certified.

4. Produce software: Relevant infrastructure software such as operating system, drivers, and application are produced and validated.

5. Installation, initialization and certification of software: It must be ensured, that software installation and initialization has occurred properly, has not been manipulated, and that security certification does indeed cover all relevant aspects.

6. Define location, valid temperature, etc.: Certify reference measurement values for calibrated devices.

7. Generate and certify signing keys: Since the scheme described above relies heavily on cryptography, and therefore on secure generation, distribution and storage of keys, these processes require verification and certification. Because of the range of possible use cases, it is difficult to find and recommend one single algorithm.

8. Specify parameter ranges: Parameter ranges for correct use of the system must be established and then, either the occurrence of lower or higher temperatures prevented, or the infrastructure design changed to avoid problems. As an example, perhaps temperature control could be included in the device in order to satisfy temperature requirements.

9. Installation of device: The installation an initialization process is critical as this is the phase where keys can be generated and exchanged.

10. Establish communication with server: The establishment of client server communication is in principle well-understood. However, there is no efficient solution currently for binding SSL keys to underlying attestation values and also the platform the key owner claims it belongs to.

11. Reference measurement record: For attestation to make any sense, reference values for the correct state of the device must be established in order to check for manipulation.

12. Document and store reference records and transfer to server: In addition to reference methods, it can also be useful to store a number of data records on the server side in order to enable sanity checks.

13. Start the boot process and time synchronization. The conditions to begin operation have been met.

14. Evidence collection: finally, sensor data can become data records that potentially can become evidence. For this reason, data records are time-stamped using the TPM.

3.3 A high-level architecture for collecting secure digital evidence

The previous section has introduced a notion of secure digital chains of evidence. Before identifying possible building blocks for actually realising the creation and collection of data for such secure digital chains of evidence, this section introduces a high-level architecture for the collection of secure digital evidence.

Obviously, it is infeasible for many real-life systems to identify, create, collect and store all possible or potentially useful digital chains of evidence explicitly. It is more feasible to identify critical events to be documented together with parameters linking events. Thus, the goal of a pro-active collection of digital evidence should be to create and store a graph of linked secure evidence records in a way that a path through the graph can represent a secure chain of evidence. Note that not all paths will represent a useful chain of evidence. Such a system as depicted in Figure 3.1 consists of the following elements:

- *Evidence generators* create data records and securely bind them to relevant parameters e.g. by digital signatures using hardware-based security [91].
- *Evidence collectors* can add semantic information to the evidence record and make it available for distribution and storage [89, 86].
- A *Forensic data-base* stores all secure evidence records as a graph structure representing the links between different events.
- Actual creation of chains of evidence is an interactive process using the *Interactive forensic data-base explorer*.

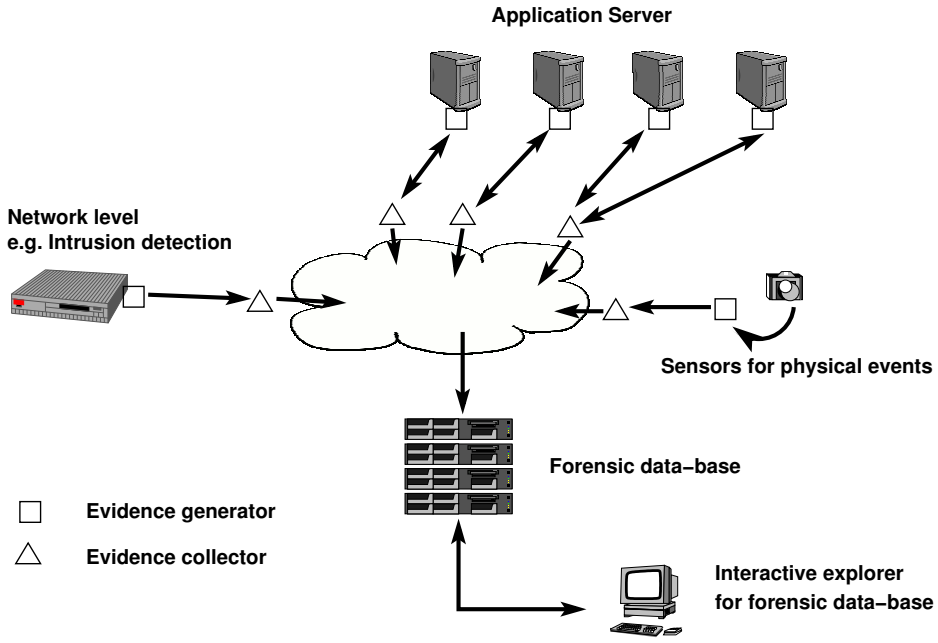

Figure 3.1: High-level architecture for collecting secure digital evidence

3.4 Building-blocks for secure evidence generation

In [91] an approach for the generation of individual secure evidence records was presented. This approach is based on established hardware-based security mechanisms and is applicable to special devices producing data records with possible forensic use. The architecture presented includes a sketch of the

process needed to ensure the security of the evidence record. Figure 3.2 shows the different steps of this process.

Even in individual data records (e.g. images taken by a digital camera), it becomes clear that the security of the collected evidence records depends on a number of steps in the process that also need to be documented. Thus, even in this relatively simple scenario, a number of events can produce additional digital evidence, thereby creating a digital chain of evidence consisting of evidential data representing events of very different types. The following paragraph introduces several building blocks that can be used to build a system supporting the construction of secure digital chains of evidence. The roles of the different building blocks correspond to the different components of the architecture for collecting secure digital evidence.

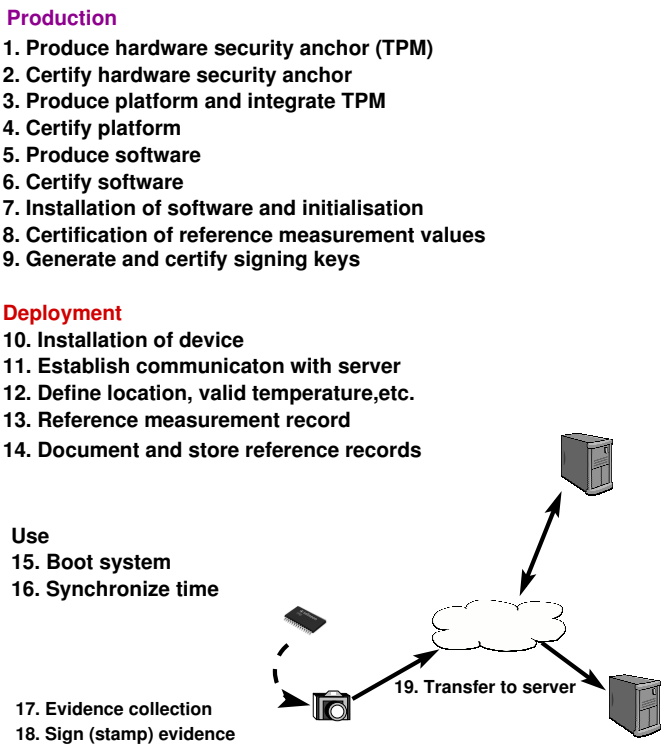

Figure 3.2: Process to establish secure evidence records

3.4.1 Secure evidence generator using Trusted Computing technology

The core part of the architecture is the actual generation of secure digital evidence. One possible approach is the use of hardware-based security mechanisms in particular Trusted Computing and the Trusted Platform Module (TPM) as specified by the Trusted Computing Group (TCG). A TPM provides a variety of security functionality. For secure evidence generation, those parts of the TPM that identify the device, bind data to the identity of the device, and provide authentic reports on the current state of the device are essential. In the context of digital cameras, the feasibility of the use of TPMs for the protection of digital images has already been proposed [91] and demonstrated [110]. The following paragraphs revise the most important parameters to be secured.

Proof of software and configuration

One important aspect of the generation of digital evidence is the status of the device used in the process. The software and configuration used to produce evidence needs to be presented and linked to the individual record. One simple scheme hereby is to include software name and version number as a simple string of text in each evidence record. This first (and often used) approach allows for uncertainties with respect to updates and various attacks on the evidence records. Just naming the software is not sufficient if the device can be manipulated. Stronger means of protection are therefore required to reliably document the software and configuration of the particular evidence generator.

To provide proof on the actual state of the evidence generator, trustworthy reporting in the device is required. The Trusted Computing standard introduces a core root of trust for measurement which establishes the foundation to report on the status by creating a chain of trust [53]. This chain of trust can be reported to external entities to allow for a verification of the evidence generator. This verification process is called Remote Attestation.

Application of remote attestation allows for a session based or per record scheme. The session based approach relies on an initial attestation of the system and a session bound to the individual evidence generator and status. Every evidence record is then cryptographically bound to this session and therefore to a particular system state. The second per record scheme involves an attestation process for each evidence record. As in the basic remote attestation, an external random number generator is involved, and longer delays as well as higher bandwidth utilisation are to be expected. As presented in [101], more advanced schemes allowing for scalable attestation schemes are to be applied.

Lightweight Infrastructure. One important feature of the proposed incorporation of Trusted Computing is the lightweight infrastructure necessary during run-time compared with a traditional Public Key Infrastructure system. Given the assertions of the hardware, a single key will not be revealed. Therefore, it is not required to maintain certificate revocation lists and to check them before a certificate is accepted. It is also not possible to move a certain identity of an evidence generator (represented by its key) from one device to the next. These inherent features allow for typical deployment scenarios like embedded, resource constraint environments.

Evidence record order

Time is a very important parameter in the forensic evaluation of evidence records. In most cases, it is absolutely necessary to have more or less precise but reliable information on when a particular event, such as the generation of an evidence record, has happened. Therefore, evidence generators need to bind evidence records to timing information. For digital chains of evidence representing a particular process this time information is essential to reconstruct the order of events in the process.

In the case of a single device, a monotonic counter (e.g. a clock) can be used to issue a time stamp for each record to ensure the order. To ensure the probative force of the time stamp, the time needs a cryptographically strong binding to the evidence record; further, it needs to be bound to time information to be issued by a trustworthy time authority. Especially the latter proves to be a strong requirement and is mostly solved by trusted third parties producing time stamps in specially secured and certified installations. However, direct online time-stamping by a trusted time source is far too inefficient for most reasonable evidence generators. Particularly in the case of embedded systems and/or high numbers of records, such a remote time stamping would create a bottleneck. Thus, a secure evidence generator

should be able to produce time stamps on its own. Of course, the remote and probably more reliable time-stamping service can be used to synchronise the local time of the evidence generator with an official time source.

A feasible approach is to introduce a certified monotonic and timed tick-counter and a mechanism for digital signatures and secure key storage to provide for time-stamps. The tick-counter as well as the cryptographic functionality should be protected by hardware-means in order to prevent manipulations by malicious software. To achieve a trustworthy local time stamp authority, the hardware being protected has to provide for a shielded monotonic counter incremented in a certain interval. To ensure this particular interval, a monitoring of the accuracy of the external clock of the hardware incrementing the counter is also required. Such techniques are available in many standard PC architectures equipped with a Trusted Platform Module (TPM) and can also be provided via Smart Cards [100]. The TPM identifies each session starting with the power up of the device with a new random number created within the TPM and is then able to time stamp arbitrary data. These time stamps can then be used to identify the order of the evidence records generated.

Considering distributed evidence generators, it is required to establish a link between the individual monotonic counters. Linking two counters results in a measure to translate between the respective local counter value into the other, which can be denoted as $n_1 := n_2 + -\mathit{offset}$. The offset is the expected uncertainty in the association due to delays on the network and computational overhead. Figure 3.3 depicts a scheme to associate one counter to the other. Hereby generator g_1 sends to g_2 a tick stamp on a random value TS. TS is then tick stamped by g_2 and sent back to g_1. The returned $stamp_{g_2}(TS)$ is then again stamped by g_1 and the resulting evidence is stored. Due to differences between the initial stamp of g_1 and the latter one, the maximum offset can be calculated and an attack on the response time of g_2 can be recorded and documented. To extend this scheme to a mutual link, the stamped result of g_2 is to be sent back to g_2. g_2, then it can stamp the actual message received to document the delay between g_1 and g_2.

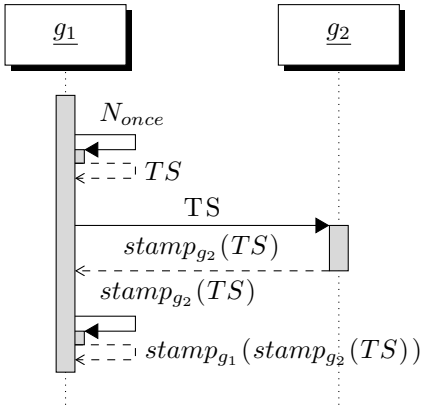

Figure 3.3: Trustworthy counter linking

Depending on the particular infrastructure, it can be necessary to link the time between several nodes belonging to one process. It can be efficient to use one central node to establish bilateral links between each node and the one central node; by doing this indirectly, all timing ticks of all nodes can be linked. Nevertheless, in highly distributed systems it can also be necessary to establish a peer-to-peer structure without any central node. In these cases, more intelligent management procedures need to be set-up in order to ensure that all events in a process can be ordered by their time tick information. Such an approach

is particularly important in networks with critical functionality as, e.g., industry controls systems used for example in the dam scenario. In such networks the synchronisation of time ticks can be combined with other existing security mechanisms distributed [69].

Real time association

The previous paragraph described that it must be possible to associate the correct order of events represented in a digital chain of evidence. For this requirement, it is sufficient to know the time an event happened in relation to other events. Another stronger requirement for valid evidence can be to know the real time an event has happened. In principle, real time information can be established in the same way. However, synchronisation can be quite loose when only the order of events is important. For real time associations there are two major differences:

- First, the time synchronisation needs to be between the evidence generator and a reliable time source, such as a certified time-stamping service. Indirect synchronisation via other nodes increases the delay and thus the inaccuracy of the time synchronisation.
- Second, the accuracy of the time synchronisation becomes relevant.

Several protocols have been proposed for the use of the TPM tick counter to represent real time information [91, 110] and also the general properties of time synchronisation protocols and algorithms have been analysed [52]. Common to all approaches is that within the digital chain of evidence also information on the time synchronisation has to be recorded. This information contains the original time stamp of the time authority but also information on the accuracy of the synchronisation, the time intervals associated with the tick counter in the evidence generator and also information to keep track of resets of the tick counter. The TPM provides support for all these parameters. One example is the tick counter in the TPM that comes with a tick nonce that identifies tick counter sessions. Tick stamps with the same nonce belong to the same session without a reset of the counter. Thus, once the tick nonce has changed, a new synchronisation with the authentic time source is necessary. It should be noted that in contrast to the proposed use of the tick counter in [110], the change of the tick nonce cannot reliably identify a re-boot of the device. As long as the TPM has power, the tick counter will not be explicitly reset during a re-boot of the device.

Other parameters

Various other parameters can become relevant for forensic use of data records. However, not all of them are readily available and can be easily or efficiently included in the secure digital chain of evidence. As an example, we briefly discuss the geographical location of the device at the time of evidence generation. Different techniques exist to determine the location. Depending on the technology used, the accuracy of the location information differs. More and more devices support the Global Position System (GPS). If adequate GPS signals are available, the GPS localisations can be in the range of 5 meters for consumer-grade GPS devices under open skies. The results within buildings under trees or with other obstacles range from 10 meters to no position information at all [108]. Other approaches, such as triangulation in wireless LAN or localisation within the GSM network are usually less accurate although they can be quite accurate in special scenarios, such as indoor localisation [59]. Parameters can have very different

characteristics; and as a means of supporting digital chains of evidence in developing systems, one has to make sure all relevant parameters are covered and maybe additional sensors are installed to enable the collection of these parameters. Examples can include the temperature of the device (very low or high temperatures can lead to corrupted evidence data), the orientation of a camera, or the names of users currently active on a multi-user device. Determining relevant parameters and their validity ranges as well as their meaning for the chain of evidence is a very important step in the engineering of such systems.

Evidence records

In addition to the different parameters related to the evidence records, it is also important to generate and secure the evidence records themselves. Obvious security measures such as digitally signing the evidence records and binding these signatures to identity the other parameters described above can be used to guarantee the authenticity and integrity of the data records in a way that these security properties are not violated by distributing evidence records and storing them in the forensic data-base. However, there can also be additional security requirements. One important factor is privacy. Evidence records can potentially contain information in individual persons or other secret information, e.g. that which is business related. Therefore, the confidentiality of evidence records shall not be neglected either. Suitable encryption should be used and other best practises for dealing with confidential data shall be applied.

3.4.2 Event collection and event correlation

In the majority of application scenarios, a certain decision by the system is not based on a singular event but on the correlation of several factors defining a certain high-level event which is defined by a set of low-level events and a process for the correlation of the low level events. Low level events are simple occurrences, like fire wall incidents and configuration changes as well as photographic evidence from the speeding camera. The process defines how these events have to interact based on an operational model where a certain high level event is to be produced.

In non-complex use cases, as presented in digital cameras [110], the evidence is generated by a single measurement agent and only the evidence records of the particular device are required. To achieve a certain probative force also in complex scenarios, it is also required to provide data on other aspects of the IT system not directly related to the evidence in question but documenting the trustworthy state of the infrastructure. For scalability reasons in bandwidth or computationally restricted applications, it is also required to split one event into a set of corresponding events.

The task of correlating events in order to construct digital chains of evidence is closely related to the task of security information and event management (SIEM) in IT networks. In particular, correlation of events from different levels and contexts, it is still very difficult in the SIEM context. In SIEM systems the correlations need to be explored at run-time to be able to induce appropriate reactions on misbehaviour. The situation is different for forensic use. Digital chains of evidence usually don't have to be created at run-time. In the case of the forensic use of event information, it is sufficient to collect the event information and maybe add additional semantic information at run-time. The actual evaluation of the correlations between events in order to produce a chain of evidence only occurs in the case of disputes or other forensic evaluations. For forensic use, a bigger effort therefore needs to be made in carefully choosing event information to be stored and in defining parameter related events.

To correlate data it is required to provide for an infrastructure supporting the processing of events from various sources in a unified structure representing the relations between the individual events.

3.4.3 Forensic data-base

For the forensic data-base, two main characteristics can be identified:

- The data-base can potentially contain huge numbers of more or less related evidence records representing graph structures where paths through the graphs can be chains of evidence also using semantic information on the events. Thus, the data-base needs to be scalable and it needs to support the exploration of large graphs with semantically enriched information.
- Evidence records need to be securely stored for a potentially long time. Storage of evidence needs to comply to regulations for long-term archiving.

For the first characteristics, the so-called triplestore [1] seems to be particularly suitable. Evidence records can consist of relatively short statements about what has happened. Triplestore is a special purpose database type developed for the use in semantic web frameworks. For a system supporting the proactive generation of secure digital chains of evidence, semantics of evidence records in terms of events that happened need to be known already at design time. The resulting structure is very similar to what can be expressed as resource triples within the Resource Description Framework (RDF) specified by the W3C (http://www.w3.org/RDF/). Some triplestore databases are very powerful with support for billions of triples loaded at a speed of more than 1.000 triples per second. Further, they support a variety of graph representations and rule-based exploration.

In addition, a variety of solutions exist for the area of secure long-term archiving. According to national regulations regarding long-term aspects of the probative force of a certain evidence, record archiving is the last step in the creation process. During the time of an evidence record in the archive, the cryptographic means used can wear out, resulting in a decreased level of trust in a specific evidence record. Existing work (e.g. [65]) shows approaches to maintaining the probative force of digital evidence in long term archives. There are also products on the market supporting long-term archiving and re-signing archived data records.

However, combining long-term archiving with a high-speed triplestore without losing the advantages of the triplestore seems to be very difficult. Therefore, it is probably necessary to follow a dual strategy for the forensic data-base where the long-term archiving and the triplestore are not fully integrated. All evidence records will go into the triple-store, but probably only a small subset really requires secure long-term storage. During the creation of evidence records, the record has to be marked in a way that the forensic data-base can decide whether long-term archiving is necessary or not. Then, a digital chain of evidence can be created using the triplestore and after completing this step long-term secured representations of the evidence records are retrieved from the long-term archive in order to produce the complete, secure digital chain of evidence.

3.4.4 Exploring the forensic data-base

This final part of the architecture for secure digital chains of evidence strongly depends on the format and data model of the forensic data-base. If the data-base is implemented as a triplestore with a good

[1] http://en.wikipedia.org/wiki/Triplestore

meta model for evidence records, a variety of tools and languages, such as Jena [2], SWI-Prolog [3] and AllegroGraph [4], can be used to develop interactive tools to explore the data-base. Relations between evidence records (and thus between events) can be graphically visualised, queries can be used to find matching evidence records or Complex Event Processing can be used to search for evidence records belonging to a particular chain of evidence.

3.5 Trusted MASSIF Information Agent

The usefulness of monitoring large systems clearly depends on the observer's level of confidence in the correctness of the available monitoring data. In order to achieve that confidence, network security measures and provisions against technical faults are not enough. As stated above, unrevealed manipulation of monitoring equipment can lead to serious consequences. In order to improve the coverage of this type of requirements in a SIEM framework, we now describe a concept and a prototypical implementation of a Trusted MASSIF Information Agent (T-MIA) [34].

3.5.1 Trust anchor and architecture

The protection of the identity of the device for measurement collection is necessary. Furthermore, the lack of control on the physical access to the sensor node induces strong requirements on the protection level.

By a suitable combination of hardware- and software-level protection techniques any manipulations of a sensor have to be revealed. In addition to the node-level protection, network security measures are needed in order to achieve specification-conformant behaviour of the sensor network, e.g., secure communication channels that protect data against tampering. This work is not intended to discuss network security, neither protection of hardware components. We rather concentrate on the important problem of clandestine manipulations of the sensor software.

A commonly used technique to reveal manipulation of a software component is software measurement: Each component is considered as a byte sequence and thus can be measured by computing a hash value, which is subsequently compared to the component's reference value. The component is authentic, if and only if both values are identical. Obviously, such measurements make no sense if the measuring component or the reference values are manipulated themselves. A common solution is to establish a chain of trust: In a layered architecture, each layer is responsible for computing the checksums of the components in the next upper layer. At the very bottom of this chain a dedicated security hardware chip takes the role of the trust anchor or "root of trust".

Trusted Computing [78] offers such a hardware root of trust providing certain security functionalities, which can be used to reveal malicious manipulations of the sensors in the field. Trusted Computing technology standards provide methods for reliably checking a system's integrity and identifying anomalous and/or unwanted characteristics. A trusted system in this sense is build on top of a Trusted Platform Module (TPM) as specified by the Trusted Computing Group (TCG). A TPM is hardened against physical attacks and equipped with several cryptographic capabilities like strong encryption and digital signatures. TPMs have been proven to be much less susceptible to attacks than corresponding software-only

[2] http://jena.apache.org/index.html
[3] http://www.swi-prolog.org/pldoc/package/semweb.html
[4] http://www.franz.com/agraph/allegrograph/

solutions.

The key concept of Trusted Computing is the extension of trust from the TPM to further system components [55]. This concept is commonly used to ensure that a system is and remains in a predictable and trustworthy state and thus produces authentic results. As described above, each layer of the chain checks the integrity of the next upper layer's programs, libraries, etc. On a PC, for example, the TPM has to check the BIOS before giving the control of the boot-process to it. The BIOS then has to verify the operating system kernel, which in turn is responsible for the measurement of the next level. Actually, a reliable and practically useful implementation for PCs and systems of similar or higher complexity is not yet feasible. Sensoring and measuring devices, however, typically have a considerably more primitive architecture than PCs and are well-suited for this kind of integrity check concept. Even for modern sensor-equipped smartphones, able to act as event detectors, but having the same magnitude of computing power that PCs had a few years ago, an implementation of the presented concept is possible. A prototypical implementation is presented in more detail now.

3.5.2 Proof of concept: Base Measure Aquisition

Figure 3.4 depicts the architecture of the T-MIA.

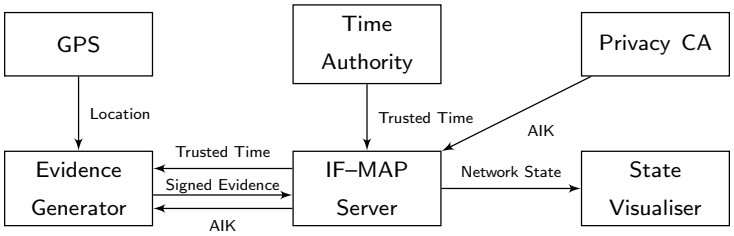

Figure 3.4: Trusted MASSIF Information Agent architecture

The main component of the T-MIA is the *evidence generator* (EG), which collects base measures and provides the measurement functions used to produce derived measures. Furthermore, the EG supports the processing of measures from external sensors, e.g., location data from a GPS module. The EG is expected to operate in unprotected environments with low physical protection and externally accessible interfaces such as wireless networks and USB access for maintenance. A necessary precondition to guarantee authenticity of the measures, is a trustworthy state of the measurement device. To meet this requirement, the EG is equipped with a TPM as trust anchor and implements a chain of trust [64]. As explained above, revelation of software manipulations is based on the comparison between the software checksums and the corresponding reference values. This comparison may be done locally within the node (self-attestation) or by a remote verifier component (remote attestation) [78].

The EG submits the collected measures digitally signed to an IF-MAP [54] server, which acts as an event information broker. During initialisation, the EG obtains two credentials from trusted third-party services for signature purposes. Figure 3.5 depicts the boot-time interaction between the EG and those services, and the role of the TPM in this interaction.

An Attestation Identity Key (AIK) is used to sign measurement results in a manner that allows verification by a remote party. The Privacy Certification Authority (PCA) issues a credential for the TPM-generated AIK. The certified AIK is, henceforth, used as an identity for this platform. According to TCG standards, AIKs cannot only be used to attest origin and authenticity of a trust measurement, but

also, to authenticate other keys and data generated by the TPM. However, the AIK functionality of a TPM is designed primarily to support remote attestation by signing the checksums of the EG's software components, while signing arbitrary data is, in fact, not directly available as a TPM operation. We have shown elsewhere, how to circumvent this limitation [63]. Hence, we are able to use TPM-signatures for arbitrary data from the EG's sensors.

Any TPM is equipped with an accurate timer. Each event signature includes the current timer value. However, the TPM timer is a relative counter, not associated to an absolute time. A *time authority* (TA) issues a certificate about the correspondence between a TPM timestamp (tickstamp) and the absolute time. The combination of tickstamp and TA-certificate can be used as a trusted timestamp. Alternatively, another trusted time source, such as GPS, could have been used.

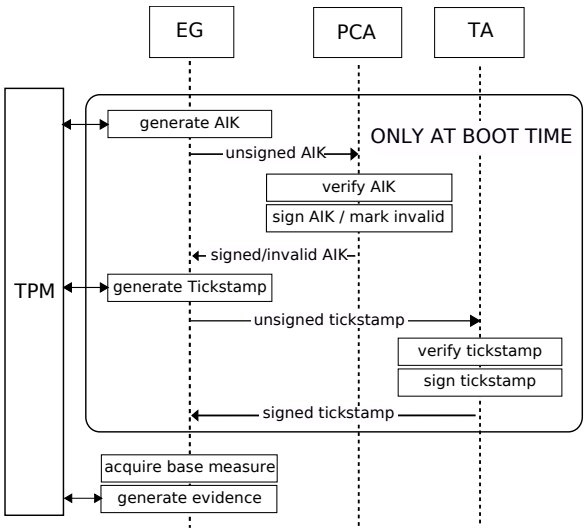

Figure 3.5: Process model

Putting it all together, a measurement record includes arbitrary sensor data, a TA-certified time stamp, and a hash value of the EG's software components. The record itself is signed by the TA-certified AIK.

Figure 3.6 shows a prototype EG, which has been implemented based on the Android smartphone platform. This platform has been selected for various reasons. Modern smartphones are equipped with a variety of sensors such as GPS, gyro sensor, electronic compass, proximity sensor, accelerometer, barometer, and ambient light sensor. Furthermore, photos, video and sound can be regarded and processed as event data. Moreover, Android is well-suited as a software platform for future embedded devices.

The TPM-anchored chain of trust is extended to the linux system and linux application layers by using the Integrity Measurement Architecture (IMA), which is integrated into any stock linux kernel as a kernel module. The Android application layer is based on libraries and the Dalvik Virtual Machine (VM). While the linux kernel layer can check the Android system libraries and the VM, Android applications run on top of the VM and are invisible to the kernel. Thus, we built a modified VM, which extends the chain of trust to the Android application level by computing the applications' checksums. A timestamp-based variant of remote attestation provided by the TPM is used for the verification of the node authenticity. All communication is based on the Trusted Network Connect (TNC) [54] protocol suite, which offers advanced security features, such as dedicated access control mechanisms for TPM-equipped nodes.

3.5.3 Use of a Trusted MIA in MASSIF

From the architectural perspective, the T-MIA implements a specific MASSIF Information Agent (MIA). A MIA is a software appliance residing in edge payload nodes. A MIA in MASSIF terminology [26] implements a remote smart sensor, that is, a MASSIF compliant sensor which allows part of the data layer functions to be performed in the payload machinery. This requires payload nodes to offer a local API to the basic sensing apparatus (syslogs, event services, etc.), and be open to installing external software modules, but apart from that, it should require minimal host modifications, allowing swift integration of MASSIF functionality into non-closed payload nodes.

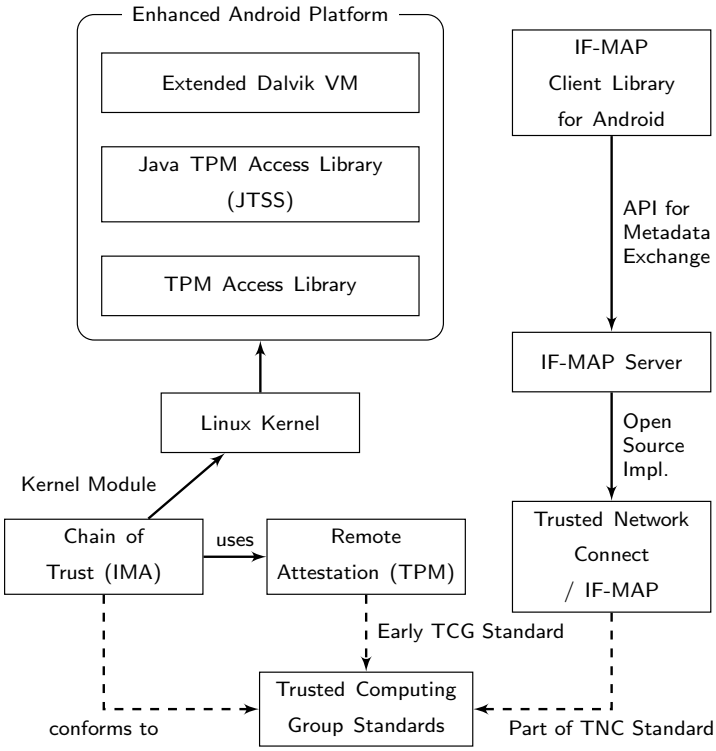

Figure 3.6: Technical building blocks

Figure 3.7 depicts a screenshot of a prototype of a mobile inspector that has been implemented by the MASSIF user group at the Technische Hochschule Mittelhessen (THM) led by Prof. Michael Jäger (THM) and Nicolai Kuntze (Fraunhofer).

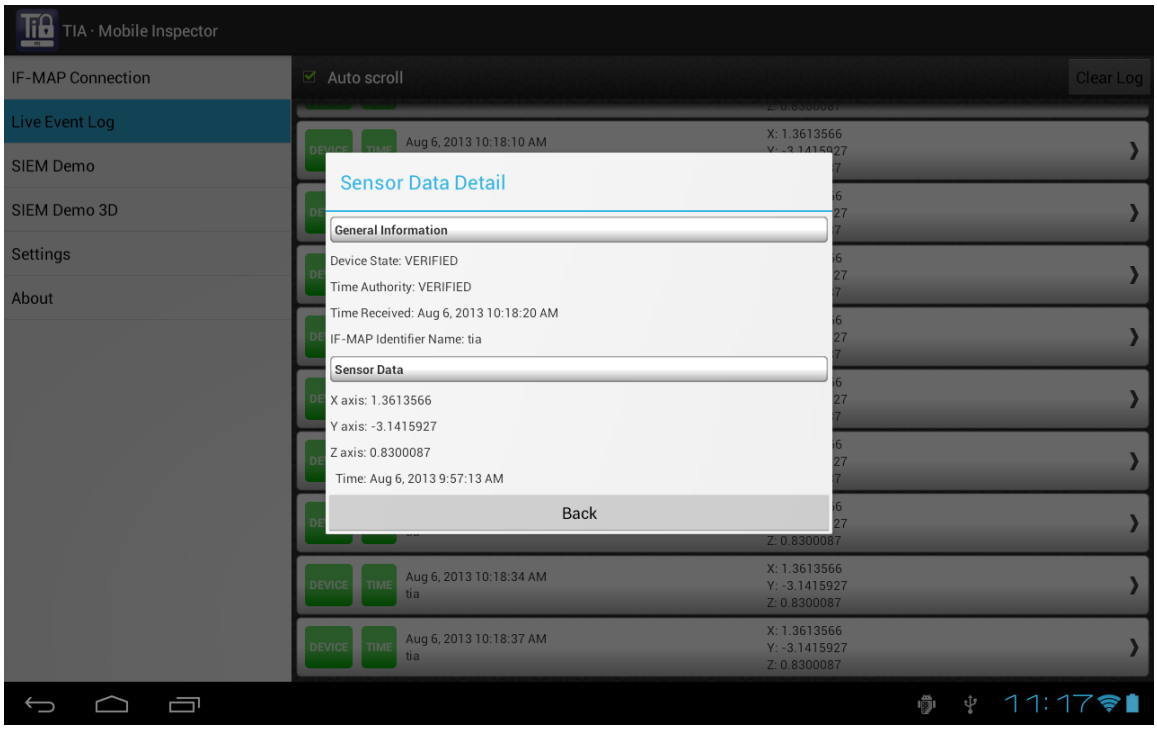

Figure 3.7: Mobile inspector - sceenshot by Michael Eckel (THM)

4 Event Reporting with Resilient Correlation Rules

This chapter presents an approach to improve the resilience of correlation rules with regard to failures that might compromise event generation and transmission at the edges of the SIEM. The approach provides a complementary solution to the mechanisms described in Chapter 3, since it can contribute to ensure correct event correlation when the collection process suffers different sorts of problems.

Improving the resilience of correlation rules is crucial to guarantee that all relevant information is collected and its integrity is maintained, but also a stepping stone to the ultimate goal of acquiring a security monitoring capability that can guide the security team through the analysis of ongoing attacks. Furthermore, it can help to increase the effectiveness of the SIEM, by making information available and decreasing the response time by triggering relevant alarms as the events occur. To achieve this objective, it is vital to perform correlation using the various data included in the collected events, thus taking advantage of the inherent redundancy in thousands/millions of events that are processed.

Correlation rules are at the core of the SIEM operation, which makes their definition an important part of the SIEM implementation, contributing to prevent attacks from circumventing the triggering of alarms as well as the possibility of an attack to go by unnoticed. Our goal is to improve current implementations of SIEM rules by expanding their resilience against attacks, even in the presence of compromised sensors (or other edge components), capable of interrupting, delaying or forging the event flow to the SIEM.

To make our approach more concrete, in the rest of the chapter we will examine some example correlation rules. These rules are built using the syntax inspired on the one used by common SIEM systems.

4.1 Elementary correlation rules

The most straightforward purpose of collected events is to raise alarms based on their content, using events from each source to define specific trigger conditions. For instance, one could define types of events that should never be observed, since they are contrary to the defined security policy or, more commonly, trigger an alarm when an event is detected more than a predetermined number of times in quick succession. Throughout the section we give examples of these out-of-the-box rules that comprise what can be considered as the current status of SIEM correlation rules, while also pointing out their frailties and limitations.

4.1.1 Rules using a single event source

Each correlation rule starts by stating the frequency parameters that should trigger an alarm. In Rule 1, we show a policy violation that should trigger an alarm to the security team, even if it happens only once

(see Figure 4.1). The policy states that all changes to user accounts must be performed using the Identity Management system (IdM), which means that if there is any change not originating from that system an alarm should be triggered.

In line 1, a time constrain is defined to trigger the rule (a mandatory field), and it states that an alarm should be triggered by the first event meeting the criteria in the remaining lines. The criterion for triggering the rule is a conjunction of three conditions. Line 2 expresses that the attacker username is different from the account used by the IdM, line 3 matches the type of event to the known category of authentication and, lastly, line 4 indicates that the outcome of the event was successful. The entire rule can be read as such: match any successful authentication events that were not executed by the IdM account.

```
1  Matching 1 events in 1 Minute with conditions(
2  NE(event1.attackerUserName,IdMAccount);And;
3  EQ(event1.categoryBehavior,/Authentication/Add);And;
4  EQ(event1.categoryOutcome,/Success))
```

Figure 4.1: Rule 1 - User changes outside IdM.

This rule relies on events from a single source, the enterprise user directory, by scanning the logs to discover change commands of a specific type and then verifying its origin based on the username. The fact that the rule depends solely on the username to determine if the change is authorized means that spoofing that information may cause attacks to go unnoticed, as long as the attacker knows the IdM username and is able to impersonate it.

The example in Rule 2 is a bit more complex, since it resorts to auxiliary rules to label events, already identifying them as attacks or successful operations (see Figure 4.2). The objective is to determine if a brute force attack was successful.

Once again line 1 indicates the time conditions, triggering the rule at each occurrence. Line 2 states that the successful login must have occurred at the same time or after the brute force attempts, with lines 3 to 5 verifying that the origin and destination of the events are the same. Line 6 enforces a white list of trusted actors, meaning that if the source of the event is in that list, the rule is not triggered. Lines 7 to 9 match the type of event and their successful outcome. The resulting rule is: match any occurrences of brute force attacks being followed by a successful authentication from the same source, provided that source is not in the trusted actors list.

```
1  Matching 1 events in 1 Minutes with conditions(
2  LE(Brute_Force.endTime,Login_Success.endTime);And;
3  EQ(Brute_Force.targetAddress,Login_Success.targetAddress);And;
4  EQ(Brute_Force.attackerAddress,Login_Success.attackerAddress);And;
5  EQ(Brute_Force.targetUserId,Login_Success.targetUserId);And;
6  "Not" InActiveList(Brute_Force.attackerAddress, Trusted List);And;
7  EQ(Brute_Force.categoryOutcome,/Success);And;
8  EQ(Login_Success.categoryBehavior,/Authentication/Verify);And;
9  EQ(Login_Success.categoryOutcome,/Success))
```

Figure 4.2: Rule 2 - Probable successful brute force attack.

Like in the first example, this rule is based on an analysis of events from a single source, an authentication server. A set of events is previously analyzed using Rule 3 and classified as a brute force attack

(see Figure 4.3). This rule then uses that information and relates it to successful authentication events to determine if the attacker achieved its goal.

```
1  Matching 5 events in 2 Minute with conditions(
2  "Not" InActiveList(Auth_Fail.attackerAddress, Trusted List);And;
3  Is(Auth_Fail.attackerAddress,NOT NULL);And;
4  Is(Auth_Fail.attackerZone,NOT NULL);And;
5  EQ(Auth_Fail.categoryBehavior,/Authentication/Verify);And
6  NE(Auth_Fail.categoryOutcome,/Success))
```

Figure 4.3: Rule 3 - Brute force logins.

In this case the time constraint in line 1 indicates that the rule is triggered only in the case 5 events meeting the criteria occur within 2 minutes. Line 2 exempts trusted actors and lines 3 and 4 make sure that the event source is identifiable, ruling out events without any information regarding their origin. Finally, lines 5 and 6 refer to the type of event and the unsuccessful outcome. Thus, the rule translates into: match 5 unsuccessful authentication attempts within 2 minutes, originating from an identifiable source that is not in the trusted list.

Once again there are clear limitations in this rule, for instance the fact that it is only triggered if the address of both the attacker and the target system are the same in the brute force attack and on the successful authentication. If an attacker is aware of this reasoning, he can use the several authentication servers normally present in a large enterprise to scatter the attack, keeping within the time limitation boundaries to avoid being detected.

To be more effective, the rule should consider the addresses of all authentication servers, although even then the attacker could still spoof its own address at each try to mask the true origin of the events. To cope with those more advanced attacks, more sophisticated rules are necessary, as we will demonstrate.

4.1.2 Rules with time based triggers

While Rule 3 took under consideration not only the attacker and target address but also the time interval between events, there are simpler rules that classify events as suspicious or even trigger alarms based solely on timing considerations.

```
1  Matching 1 events in 1 Hours with conditions(
2  EQ(event1.deviceEventClassId,Security:630);And;
3  InActiveList(event1.targetUserId, CreatedAccountsActiveList))
```

Figure 4.4: Rule 4 - Windows account created and deleted within 1 hour.

Rule 4 uses event information from the user directory and relies on a related rule that adds newly created user accounts to the active list (mentioned in line 3 : ”CreatedAccountsActiveList”) (see Figure 4.4). The entries added to this list have a Time to Live (TTL) of one hour, after which they are automatically removed from the active list by the SIEM. If, during that hour, the account is deleted, identified in line 2 by the event code 630 in Windows-based domain controllers, this rule would be triggered and the action could be marked as suspicious or even display an alarm to the security team, who would then proceed to review the actions performed using that account.

By not relying on relating time constraints and the source or destination addresses, this rule can be somewhat sturdier than previous examples. Nevertheless, as in all time based rules, if an attacker is aware of the restrictions imposed by such triggers, he can easily bypass the rule and consequent alarms. In this specific case, creating the bogus account and waiting one hour before using it could successfully perform the attack.

4.1.3 Some of the limitations of basic correlation rules

The security team is highly exposed to possible faults by relying on only one event source and/or in time constraints to determine if a rule should be triggered. As soon as an attacker is aware of how the correlations rules are built, the loopholes become evident, thus making a targeted attack possible.

As we showed before, basic rules are normally easy to bypass either by spoofing part of the event details, such as the username or the IP address, something that usually can be done without much effort. Likewise, triggers based on the elapsed time between events can also be bypassed if the attacker is able either to change the pace of the attack, widen the scope of targets or simply delaying the sending of event information by the sensors.

Even if the attacker is unable to compromise the components of the SIEM system, he can circumvent basic rules just by compromising the sensor collecting events from the source under attack. The only option available to minimize the number of missed alarms, when one or more sensors are compromised, is to collect information from different sources, using the inherent relation between those sources as a way of enriching the correlation rules. Both the network and asset models[1] can be very helpful when designing a robust set of correlation rules, since they contain precious information regarding the event sources, their inherent characteristics, location in the network and communication channels between them.

4.2 Improving correlation rules

As we have demonstrated in the previous section, standard correlation rules can be ineffective against even moderately sophisticated attackers and are unable to cope with either accidental or malicious faults, such as compromised sensors. Our goal is to eliminate as many frailties in the correlations rules as possible, improving their resilience without adding any more complexity than strictly necessary.

Much like when improving the security in the configuration of a system, the first step should be to *harden the correlation rule*, considering non-straight-forward scenarios even when using a single event source. Our approach is to enrich the correlation rules using further information (called *properties*) included in the events, and take advantage of SIEM resources such as the asset and network models. The default rule set takes the integrity of information for granted and focuses mostly on the best-case scenario, which results in the weaknesses mentioned earlier. Instead of considering only part of the information that constitutes an event, we take advantage of as much information as possible to detect malicious behaviors, even if the attacker is taking some precautions to go unnoticed. Furthermore, by

[1] The SIEM system can support the creation of lists containing information about the monitored infrastructure. Two of those lists are the network and asset models, which can improve the efficiency of correlation rules by being able to associate event sources and determining the relationship between them. The network model contains information about the hosts and how they are connected through the network equipment. The asset model enables the SIEM to determine if a certain asset is, for example, a web server, a router or a firewall. Both the network and asset models can be updated automatically using information from sources such as vulnerability scanners or other similar tools capable of maintaining the infrastructure inventory.

broadening the scope of properties considered when defining the rules, it is possible to increase the difficulty of forging event information.

Understanding the properties of events and their idiosyncrasies is important when designing more resilient correlation rules. A subset of those properties is common throughout events from multiple sources, such as source/attacker and destination/target addresses, the event type or the outcome of the event. These fields can be used in any rule and constitute the basis from most correlation rules. However, there are many others that are exclusive to specific technologies, making them extremely pertinent when designing resilient correlation rules. By acknowledging the specificities of event properties it is possible to broaden fault detection capabilities and deepen the level of detail that will help to improve rules.

While hardening the correlation rules allows the SIEM to detect previously unobserved abnormal actions, basing the evaluation of events in a single source keeps the system vulnerable to the successful attacks on that source. This vulnerability results in situations where an attacker, which is able to compromise a single component or system, can completely control the events being generated in that source, thus thwarting correlation rules and allowing an attack to go by unnoticed. We demonstrate that it is possible to combine information from multiple sources in order to strengthen correlation rules, making them effective even in the presence of a partially compromised infrastructure.

The idea behind correlating events from multiple sources is that all systems are interconnected, and therefore, most actions result in associated events being generated at more than one source, thus creating some level of *redundancy on the information* that reaches the SIEM engine. Let us consider that an attacker is able to compromise a server without being detected, and subsequently disables event collection from that source. If the attacker starts to use that compromised server to launch an new attack, each communication made with other servers will generate events on those destination servers, as well as in the network components that connect both assets. Therefore, even in the presence of a compromised source, it is possible to collect events from other sources that convey the information needed to detect an attack. Taking advantage of these associated events in different sources, it is possible to design resilient correlation rules that not only increase the effectiveness of attack detection but also allow the security team to identify possibly compromised systems.

Even more interesting is to utilize events from multiple sources not only to detect but also to mask faults. As we exemplified earlier, some actions are expected to generate events both in the source and destination systems, therefore incoherencies between those sources are sufficient to raise an alarm, detecting a possible fault. However, in the case of actions that generate events in more that two sources it may be possible to go further, for instance by employing a voting mechanism to determine which of the sources is reporting incoherent information. The remaining sources can then be used to discover the ongoing attack, so that the invalid source can be identified as reporting incoherent information.

4.2.1 A method for improving correlation rules

The step-by-step improvement of correlation rules can be performed accordingly to the methodology outlined in Figure 4.5. Depending on the type of rule and event sources, some of the steps of the methodology may not apply. There are exceptions to every rule, and in this case we opted for a generic approach that fits most cases, adapting it for specific situations when necessary.

To remove existing liabilities in the original, less resilient, rule, it is necessary to identify ”blind spots” or possible limitations in the conditions of the rule. Having a whitelist or other exception mechanism is undesirable, unless strictly necessary, as it allows an attacker to circumvent detection by impersonating a trusted actor, sometimes just by forging the source IP address. There are situations where

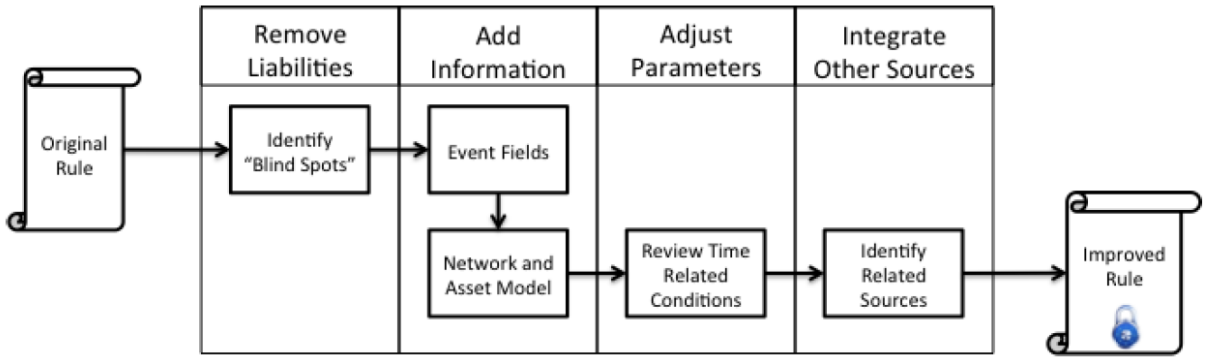

Figure 4.5: Method for improving the resilience of a correlation rule.

the usage of whitelists is justifiable and necessary, for instance when considering specific systems that perform otherwise forbidden tasks like vulnerability scanners. However, in those situations, it is vital to carefully define the exceptions, providing comprehensive information not limited to IP addresses or hostnames.

The addition of relevant information is twofold, with the first and most simple approach being to take advantage of unutilized fields in the events. Information, such as the hostname, port or the agent that collected the event, can present valuable insight when analyzing events, increasing the necessary knowledge of the system that the attacker must possess to successfully manipulate the information entering the SIEM. The second step would be to incorporate information from the network and asset models, with the advantages we mentioned above. Since the integrity of the information in these models is verified and only a SIEM administrator can update their contents, the models can be used as baselines to compare against the event data.

The majority of correlation rules are partially based on the time lapse between events or the number of similar events within a defined time frame, and consequently these time constraints have to be carefully defined and reviewed. The definition of time intervals to consider when triggering alarms can be the difference between an attack being detected or not. On the other hand, it can also cause false positives or false negatives that decrease the confidence in the alert capabilities of the SIEM and flood the security teams with information. Both risks have to be taken into consideration when constructing the correlation rules, as they result in the loss of vital information, either because alarms are not triggered or due to an overflow of information that exceeds the processing capability. The solution must rely on a carefully designed learning process, tuning the parameters according to the normal operation of the infrastructure. Nevertheless, it is possible to define approximate default values for each scenario based on the experience of the security teams.

The final step when improving a rule would be to identify related sources that could contribute to verify the veracity of the information being processed by the SIEM, thereby making it resilient to a limited number of compromised sensors. The previous methods are effective against an attacker trying to inject bogus data in the network, and increase the overall robustness of the infrastructure, but they are not able to cope with compromised components where an attacker is able to penetrate the outer defense layer, consequently possessing all the necessary information to deceive the SIEM rules. The possible solution is to collect information from multiple related sources, increasing the resilience of the system by considering that (at least initially) an attacker is only able to compromise part of the systems.

4.2.2 Correlation rule hardening

This section applies the proposed methodology to improve the resilience of the correlation rules. We have established that Rule 1 is vulnerable to spoofing (see Figure 4.1), as the knowledge of the IdM account name enables a bypass. This attack is possible because the rule verifies only the origin account, disregarding any additional information like the source system. The rule may thus be improved by adding information from other event fields. These properties can be used to include further details from that source system, forcing a possible attacker to have to spoof more information, thereby making the attack more complex.

```
1  Matching 1 events in 1 Minute with conditions(
2  (NE(event1.attackerUserName,IdMAccount);Or;
3  NE(event1.attackerAddress,IdMAddress);Or;
4  NE(event1.attackerOS,IdMOS));And;
5  EQ(event1.categoryBehavior,/Authentication/Add);And;
6  EQ(event1.categoryOutcome,/Success))
```

Figure 4.6: Rule 5 - User changes outside IdM (improved).

Rule 5 is an improved version of Rule 1, where the underlined conditions were added in lines 3 and 4 to force additional checks (see Figure 4.6). Using the attacker address and operating system signature it is possible to make the rule more robust, forcing a possible attacker to spoof not only the account name but also the address and OS fingerprint of the IdM system.

However, to make these conditions possible one would have to configure a large set of variables in the SIEM, consequently increasing the operational efforts and configuration complexity. Fortunately, the SIEM system includes the aforementioned network and asset models, which can be automatically updated with relevant information from the infrastructure including, but not limited to, the address and OS fingerprint of the servers. The methodology proposes the use of the asset and network models to ease the management of correlation rules, adding information maintained and updated by the SIEM to better identify source or destination systems.

The hardened rule is therefore not only more powerful but also easier to manage (see Figure 4.7). Since the only source authorized to perform the action of adding a new user to the domain is the Identity Manager system, it is imperative for the source of such actions to be part of the asset model. The properties of the event source must be checked against the information present in both the asset and network models, something that can be easily enforced.

```
1  Matching 1 events in 1 Minute with conditions(
2  (NE(event1.attackerUserName,IdMAccount);Or;
3  “Not” InAssetModel(event1.asset);Or;
4  NE(event1.asset, AssetModel.IdMAsset);Or;
5  “Not” InNetworkModel(event1.attackerAddress);Or;
6  NE(event1.attackerAddress, NetworkModel.IdMAddress));And;
7  EQ(event1.categoryBehavior,/Authentication/Add);And;
8  EQ(event1.categoryOutcome,/Success))
```

Figure 4.7: Rule 6 - User changes outside IdM (hardened).

The conditions in lines 3 and 4 verify that the source system is part of the asset model and corresponds

to the asset declared as the IdM, while lines 5 and 6 focus on the network information to establish the correspondence. The resulting Rule 6 would then be able to verify not only specific event attributes but consider the properties of two objects, matching them to encounter relevant discrepancies that indicate the attack source is not the same.

It is possible to use similar improvement techniques in Rule 2 and Rule 4. In the first case, the initial step would be to consider attacks coming from diverse sources and against distributed authentication servers. If an attacker is able to compromise several computers, for instance using a worm, with the objective of performing a brute force attack against a privileged account, the worm could instruct the infected machines to perform sweeps across the multiple authentication servers, thereby avoiding the time constraints on the rule.

```
1  Matching 1 events in 1 Minutes with conditions(
2  LE(Brute_Force.endTime,Login_Success.endTime);And;
3  EQ(Brute_Force.targetAddress,Login_Success.targetAddress);And;
4  EQ(Brute_Force.attackerAddress,Login_Success.attackerAddress));And;
5  EQ(Brute_Force.targetUserId,Login_Success.targetUserId);And;
6  ("Not" InActiveList(Brute_Force.attackerAddress, Trusted List));And;
7  EQ(Brute_Force.categoryOutcome,/Success);And;
8  EQ(Login_Success.categoryBehavior,/Authentication/Verify);And;
9  EQ(Login_Success.categoryOutcome,/Success))
```

Figure 4.8: Rule 7 - Probable successful brute force attack (hardened).

The improved Rule 7 would consider the number failed authentication attempts by the same account, regardless of the origin address, followed by a successful authentication by that same account (see Figure 4.8). The original rule also included a loophole by considering trusted sources, effectively ignoring events originating from systems with addresses on that list, which could be ranges of addresses inside a trusted network perimeter, therefore creating a blind spot if the attacker is able to breach that supposedly secure zone. The elimination of whitelists that may introduce vulnerabilities is the first step of the proposed methodology to improve correlation rules. Since we are focusing in network information, the event fields used to construct and improve these rules are part of the set of properties that are common to events from all sources, without the necessity of resorting to specific properties from this event source.

Hardening Rule 4 requires additional efforts, since the simplicity of the objective would be undermined by a more complex construction, possibly increasing the number of false positives (see Figure 4.9). Our only proposal, following the improvement methodology, is to widen the time window between the creation and deletion of an account, since it is not expected a user account to be active less than 48 hours when considering the normal life cycle of domain accounts. Although studies indicate that a security breach may remain undetected on average for 416 days [60], the threshold of 48 hours seems appropriate to deal with the most eminent threats. A longer time interval or any further conditions would dramatically increase the number of false positives and the amount of information to be processed by the security team, in fact decreasing the probability of an attack being uncovered.

```
1  Matching 1 events in 48 Hours with conditions(
2  EQ(event1.deviceEventClassId,Security:630);And;
3  InActiveList(event1.targetUserId, CreatedAccountsActiveList))
```

Figure 4.9: Rule 8 - Windows account created and deleted within 48 hours.

The resulting Rule 8 employs the deviceEventClassId property of the event to determine the originating action. This property is specific to events from Windows servers, more precisely domain controllers, therefore not part of the common set shared by all events.

4.2.3 Correlating different event sources

Even after the process of hardening the basic rules several limitations are still present. As we have mentioned above, relying on a single source of events to trigger alarms is ineffective when considering a fault model where event generation might be affected. The final step of the methodology proposes the correlation of events from multiple sources, withdrawing data from separate systems or devices to increase the resilience of the process.

Validation Using Network Events: Computer networks are ubiquitous in any modern IT infrastructure, with each node being connected to one or more network components in order to communicate with applications, databases or other systems. This means that each request or command from a source system is bound to have passed by a number of network nodes before reaching its destination, enabling the correlation of events from those sources.

One of the first event sources to incorporate in a SIEM system is the firewalls due to their extensive logging of established connections, detailing traffic classification, protocol information and used ports. Using this information, as well as the defined network model, it is possible to detect attempts to mask the real origin of the traffic by spoofing the source address.

We demonstrate this capability in Rule 9, based on the already modified Rule 6, to detect changes in user accounts not performed by the authorized IdM application (see Figure 4.10). The first step would be to define a rule that processes firewall logs and identifies commands from the IdM application to the user directory server, adding those commands to an active list for one minute. The active list can only be updated using events from a specific firewall, which can be enforced by using a communication protocol between the firewall sensor and the SIEM that ensures authenticity of the origin, something that can be done effortlessly, for instance using IPSec.

```
1  Matching 1 events in 1 Minute with conditions(
2  ((NE(event1.attackerUserName,IdMAccount));Or;
3  (Not InAssetModel(event1.asset));Or;
4  NE(event1.asset, AssetModel.IdMAsset);Or;
5  (Not InNetworkModel(event1.attackerAddress));Or;
6  (Not InActiveList(event1.command, IdMCommandsInLastMinute));Or;
7  NE(event1.attackerAddress, NetworkModel.IdMAddress));And;
8  EQ(event1.categoryBehavior,/Authentication/Add);And;
9  EQ(event1.categoryOutcome,/Success))
```

Figure 4.10: Rule 9 - User changes outside IdM (using firewall events).

By stating that if one of the conditions is not met an alarm is triggered, we are eliminating the possibility of an attacker using a compromised workstation somewhere in the corporate network to impersonate the IdM server and successfully create a user account. If the attacker tried to compromise the sensor collecting the firewall events, the change in the user directory would trigger the alarm, since by blocking the events from the firewall the attacker would also hamper the update of the active list, therefore triggering the alarm all the same.

The last resort available to the attacker would be to stealthily compromise a machine in the same network zone as the IdM system, already a more secure perimeter, and only then spoof the origin of the command.

Fault Detection Using Correlation: Event correlation can be used directly to detect incoherent information from multiple sources recording the same event, as we have seen in Rule 9. By taking advantage of the network and asset models, it is possible to define not only acceptable commands, but to verify how those commands align step-by-step with defined workflows and procedures. The SIEM is able to interpret information such as the type of asset to detect abnormal behavior by analyzing the events coming from that asset.

For example, an institution might decide that some operations, like deploying firewall rules or software updates, can only be performed after working hours to avoid performance impacts. A simple rule can be employed to determine if certain types of events do not occur outside the allowed time window. However, this rule might only apply to production systems, while development or test environments have less strict policies. The asset model can be used to enrich the correlation rules with information pertaining to the infrastructure in which the systems are deployed, making it possible to accommodate these nuances.

Using the same principle the SIEM is also able to detect if specific changes to the configuration of the systems are being performed from the operations center or from the technicians personal laptops over a VPN connection. The company policy might dictate that critical operations can only be performed locally to ensure direct access to the systems in case a rollback is needed, therefore SIEM rules can be defined to detect such occurrences and trigger the necessary alarms.

Using a similar approach it is also possible to detect faults in the ICT infrastructure by spotting the absence of expected events. Suppose that an attacker decides to target a sensor attached to a web server with the objective of modifying its contents, which are in turn stored in a separate database. Unless the attacker is also able to successfully compromise the sensors in the database and in the firewall segregating the DMZ from the internal database servers, there would still be traces of the malicious actions. An alarm can be triggered upon the verification that events from the database and firewall indicate an action originating from the web server, while the associated event from that source is absent from the SIEM. The alarm would state that an expected event did not reach the SIEM, indicating a possible fault in that source or in the collection process.

With this scenario in mind, the goal would be to create pattern-based correlation rules that, once again using the network and asset model, are able to match related events therefore also detecting missing events that should have been received by the SIEM.

Fault Masking Using Correlation: More than just detecting faults, in specific situations correlation rules may go as far as permitting fault masking, which is to say that the SIEM system can reach the same conclusions and trigger alarms even in the presence of compromised components. The principle of analyzing not only single events but also entire workflows, as we described above, enables the SIEM to process complex information by relating information from multiple sources.

We have shown how to use correlation to detect faults and trigger the correspondent alarms, but let us consider a situation where a command workflow is supposed to generate events in three different components. If, after correlating the information from all sources, it is discovered that one of the events is either missing or unaligned with the remaining, the SIEM could disregard the entire flow and merely trigger an alert to the security team. But, by employing a voting algorithm, it is also possible to assume that there was either an error in the outlier source, or that it has been compromised.

4.2.4 Limitations of correlation rules to detect attacks

There are inherent limitations to detecting attacks relying only on correlation rules, as information redundancy is not always present. Let us put forward a scenario where a software component installed on top of the operating system acts as a sensor for events in that source. If an attacker is able to introduce malware in that machine, for example using an infected USB drive, then the malware could immediately target the sensor, much like well-known malware that disables the anti-virus agent. Imagining that the system is the intended target for the attack, for instance to steal information stored in the hard drive, the attacker would have no need to use the network thus making it impossible for other sensors to detect the attack. The lack of information redundancy, i.e., information coming from only one source, means that those events will not reach the SIEM in case that source is compromised.

Countering these targeted attacks typically cannot be done using the rules of a SIEM system, with the answer residing in stricter security policies like disabling USB ports. However, since in this chapter we are designing ways to improve SIEM resilience (not increasing its capabilities), we can focus on more common attacks that make use of the network to access remote systems and spread across the IT infrastructure.

Improving the resilience of correlation rules is also an exercise to increase attack and fault detection capabilities while ensuring that the rate of false positives is maintained or, preferably, improved. As correlation rules become more detailed, using specific information from the sources, and incorporate events from multiple sources, the knowledge of the infrastructure must also be on par with those information requirements. The more specific the rule, the more susceptible it is to changes in the monitored systems, meaning that updates or changes in processes can result in the necessity of reviewing the correlation rules in order to avoid erroneous alarms. For instance, when information from the asset or network model is used, one must ensure that changes to the systems are readily updated in those models, one of the options being to populate the models using automated scanning tools.

4.3 AutoRule: Automatic analysis of rules

Under certain conditions it is possible to automate the analysis of correlation rules, helping to identify the rules that should be reviewed before implementation. Taking into consideration the complexity of some correlation rules, the automatic process will have inherent limitations when compared to human reviews performed by security experts. Nonetheless, a systematic approach will enable the detection of the most common errors when constructing correlation rules, as well as pointing out improvement possibilities.

The automatic analyzer could start by parsing the rules and identifying keywords. Heuristic analysis can then be applied to pinpoint possible frailties, suggesting improvements. The proposed methodology should be followed step-by-step, firstly identifying the usage of whitelists, then the lack of event information diversity, followed by an absence of references to the network and asset models in conjunction with other event properties.

Time related conditions can be compared to standard values based on the type of rule, however, as we mentioned earlier, there should be a learning process to adjust parameters accordingly to the specific characteristics of the infrastructure. Lastly, to identify possible related sources, the tool should have the ability to import data from the asset and network models, creating an internal knowledge base capable of adding relevant event information to the correlation rules. The tool should therefore enable the possibility of customization by the security team, adapting to the monitored systems.

To demonstrate a systematic analysis of SIEM rules, following the methodology presented previously, we developed AutoRule (Automatic Rule Analysis), a proof-of-concept application in Java to parse correlation rules, suggest improvements and calculate the overall resilience score according to the verified level of redundancy. The score is estimated according to the identified shortcomings of the rule, with different weights being given to diverse occurrences and a lower score indicating a more resilient correlation rule.

The first step, as the methodology advocates, is to detect the presence of exceptions to the rule by verifying the employment of trusted lists. As we explained before, if an attacker is aware of that potential loophole, it may be possible to forge data in order for the attack to pass unnoticed by the SIEM. Being a relevant source for attack misidentification, the presence of a list of trusted nodes is translated into a significant increase of the overall score.

AutoRule also checks for network or account information used in isolation, therefore making the rules weaker. Not only is the combination of both conditions recommended, but resorting to the network and asset models instead of individually managing variables is also advised.

```
Line 5: "Not" InActiveList(Brute_Force.attackerAddress, Trusted List);And;  - Possible loophole in active list exceptions
Line 3: EQ(Brute_Force.attackerAddress,Login_Success.attackerAddress);And;  - Address reference should be complemented with account information
Warning - Network conditions should rely on the network model

Final score: 9
```

Figure 4.11: Evaluation of Rule 2 with AutoRule.

Figure 4.11 represents the output of the verification process applied to Rule 2. It immediately shows that this rule is not very resilient, considering that it includes a reference to a trusted list, identifies the originating agent solely based on the network address and makes no use of the network or asset models.

Applying the tool to Rule 5, an already improved rule, shows the differences in robustness and, consequently, in the attributed score as we can observe in Figure 4.12.

```
Warning - Network conditions should rely on the network model
Warning - Account verifications should rely on the asset model

Final score: 2
```

Figure 4.12: Evaluation of Rule 5 with AutoRule.

The final version of this rule, Rule 6, including all the recommended changes to its structure, raises no flags and therefore has a final score of 0.

5 Resilient Event Bus

This chapter presents the design, some implementation details and evaluation of the Resilient Event Bus (REB). The REB is mainly in charge of disseminating the events collected by the edge-MIS, after being pre-processed by the Generic Event Translation (GET), towards the core-MIS. The core-MIS then uses some alternative communication mechanism to forward the events to the Complex Event Processing (CEP) engine. Less often, the REB may also need to transmit messages in the opposite direction, from the core network to the edges, for instance when information needs to be conveyed to the sensors. Consequently, the REB must provide a bidirectional communication path among the MIS, with the attribute that most of the traffic goes from the edge to the core.

The REB is organized as an overlay network built among MIS nodes, which are named generically as *REB nodes*. The communication in the overlay network is performed on top of the UDP/IP protocols, allowing the support of different network settings. REB uses application-level one-hop source routing to send messages towards the destination, instead of simply following the routes imposed by the network-level routing. It also takes advantage of coding techniques and the available redundancy of the network, such as when a node has multiple network connections (i.e., multihoming), to ensure that messages arrive securely with a very high probability.

5.1 REB Overview

The communication among the MISes plays a fundamental role in the MASSIF resilience architecture. This feature is responsible for delivering events from the edge services to the core SIEM correlation engine despite the threats affecting the underlying communication network. Accordingly, the REB design is influenced by the way SIEM systems are deployed, which are usually distributed over several facilities. A facility corresponds to a subset of the overall network, where a set of sensors provide event data to a local edge-MIS (left side of Figure 5.1) or where a group of machines implement the engine (and other supporting services) connected through the core-MIS (right side of Figure 5.1). A facility can therefore be modeled as a LAN, and the associated MIS can be seen as a routing device that receives the data produced locally and forwards it towards the final destination facility. The interconnection among the facilities can be abstracted as a WAN. One however should keep in mind that LAN and WAN are modeling artifacts, since in practice they will depend on the actual deployment of the SIEM. In one extreme case, the WAN can be the Internet, if the SIEM collects information from various offices of an organization that are located in different regions (of the same country or different countries). In the other extreme case, the WAN could be a set of switches with virtual LANs that interconnect a few PC racks on a data center. This organization has the virtue that entails no meaningful modification to the existing SIEM system and only requires the introduction of a REB node in each MIS at the border of a facility.

REB nodes communicate through the UDP/IP protocols, defining an overlay network atop the IP

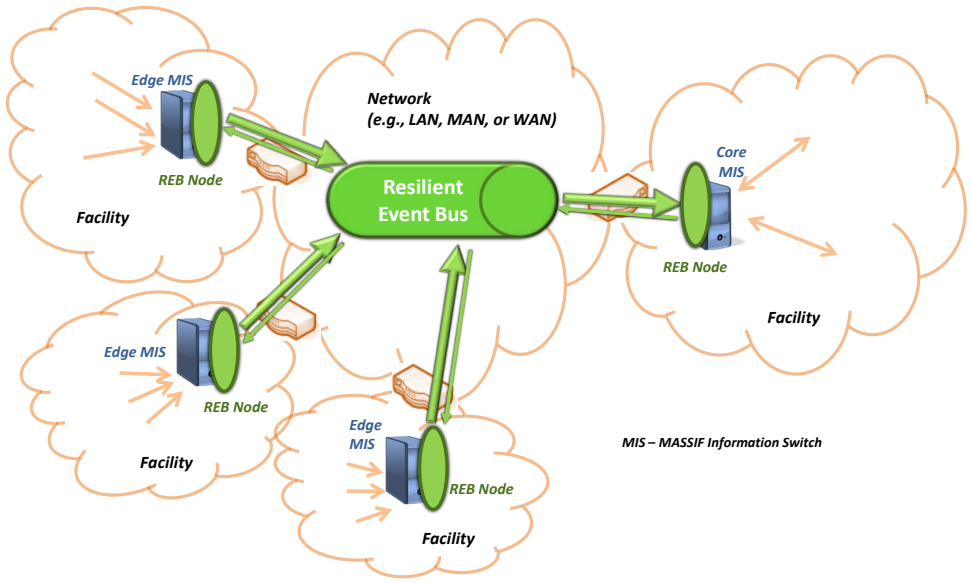

Figure 5.1: REB topological view.

network and running application-level routing strategies to select overlay channels that are (expectedly) providing correct communication. Overlay networks have been used as mechanisms to implement routing schemes that take into account specific application requirements [6]. In MASSIF we want to employ an overlay network to create redundant network-agnostic channels for efficient and robust communication, namely for event transport from the edge sensors to the core event correlation engine.

Depending on the network setting, we envision different kinds of faults that might preclude the transmission of data unless appropriate measures are enforced among the REB nodes. For instance, accidental or/and malicious faults can cause the corruption, loss, re-order and delay of packets. An adversary might try to modify the event data carried in a packet or add new events to prevent the detection of an intrusion. Congestion or denial of service (DoS) attacks can also make certain IP routers between specific REB nodes unresponsive, causing significant packet loss or the postponement of their delivery. A strong adversary might also be able to take control of one of the IP routers, allowing her the capability to perform sophisticated forms of man-in-the-middle attacks. REB nodes may also suffer failures, such as a crash or a compromise by an adversary. In this case, the REB per se will not solve the specific problem of the failed node, and consequently a facility using a REB node as a proxy might become disconnected. However, as a whole, the REB should continue to work properly, allowing the remaining nodes to continue exchanging data.

Many of the above threats can be addressed with well known security mechanisms that have been applied to standard communication protocols, such as SSL/TLS [42] and IPsec [83]. For example, each pair of REB nodes shares a symmetric secret key. Leveraging from this key, one can add to every packet a Message Authentication Code (MAC) that allows the integrity and authenticity of the arriving packets to be checked in an efficient way. Consequently, any packet that deviates from the expected, either because it was modified or was transmitted from a malicious source, can be identified and deleted. This means that insertion attacks are simply avoided, and corruptions are transformed into erasures that can be recovered with retransmissions (or in our case, also with coding techniques). Optionally, one can also encrypt the packet contents, which guarantees confidentiality and prevents eavesdropping of packet data.

Based on these mechanisms, the REB is able to reduce substantially the attack surface that might be exploited by an adversary. However, they only put the REB at the same level of resilience as standard solutions, which are insufficient to prevent certain (sometimes more advanced) attacks. For example, an adversary might thwart the correlation of two sources of events, if she is able to delay the packets that contain the events from one of the sources. This occurs because events are correlated within a predefined time window, and if they arrive in different windows the associated rule might not be activated. Alternatively, an adversary might perform a DoS in one of the IP routers that forwards the packets from one of the sources, causing very high transmission losses and the continuous retransmission of packets (to recover from the packet drops). The overall effect of the attack is once again the delay of certain event packets, or eventually the temporary disconnection of one of the sources. Traditional solutions for secure communication, such as those based on TLS over TCP, are also vulnerable to other more specific attacks [18]. For instance, this approach is fragile to an adversary with access to the channel between the two communicating parties, since she can continuously abort the establishment of any TCP connection by sending *Reset* packets, creating in fact unavailability on the communications.

This sort of problems are addressed by the REB by employing multipath transmission that exploits spatial redundancy to attain high levels of robustness, while combining it with the use of erasure codes and route monitoring techniques to maintain an efficient communication, even in the presence of attacks.

5.2 Communication properties

The REB offers point-to-point communication channels between nodes. Data is transmitted as a stream of bytes, and is delivered reliably in first-in first-out (FIFO) order. The arrival of duplicate data is identified and removed, and flow control is enforced at the senders to prevent receivers from being overwhelmed with too much information. The REB also ensures data integrity and authenticity, and confidentiality can be guaranteed on an optional basis, depending on an indication by the application.

One of the main reasons for choosing the above properties was to mimic the experience offered by SSL/TLS running on top of TCP. SIEM systems are sometimes built with this kind of communication, and therefore, we wanted to match the expectancy of the developers. Given the fact that the REB is implemented on top of UDP/IP — which offers a much weaker service — it was necessary to devise a group of mechanisms to enforce these properties.

The REB also provides a robust multipath communication that tolerates some faults in the underlying network and also in intermediary REB nodes, which would interrupt the communication if, for instance, a single TLS/TCP channel was used instead. The usage of multiple paths to transmit data from a source to a destination is done in an efficient way by taking advantage of erasure codes to minimize retransmissions, and which is further optimized by picking the paths with the best quality of service .

In the rest of this section, we explain generically how these properties are achieved.

5.2.1 Reliable delivery

Reliable delivery of data can be attained when the underlying communication network provides at least fair loss delivery, that is, if it makes some effort to transmit a packet through a route and then deliver it to the destination. The Internet is one of such networks, and using UDP on top of it does not strengthen its fair loss property, therefore it may happen that a transmitted packet is lost in-transit (e.g. due to congestion or a crash of a network router). On the other hand, TCP provides reliable delivery (assuming

both source and destination processes are correct) by segmenting the input stream and by transmitting one segment at a time, while waiting for the arrival of an acknowledgment of their reception. A retransmission occurs if it appears that the segment was lost, e.g., when the retransmission timer expires at the sender. As each segment is received and acknowledged, the receiver delivers them to the application.

The REB employs a similar strategy to provide reliable delivery. The input stream from the application is saved in a queue, and then split into several segments for dissemination. Unlike TCP, however, retransmissions based on timers are avoided when possible because we employ spatial redundancy (transmitting through multiple routes) to provide the necessary robustness. To minimize the message overhead of the redundancy, the REB also preprocesses each segment with an encoder that applies an erasure code — a kind of Forward Error Correction (FEC) code — to produce a number of packets. Depending on the code that is applied, if it is systematic or not, the resulting packets may contain the original data plus some repair information, or they may just have encoded data (see Section 5.5.5). The overall sum of the packets lengths is typically larger than the original segment, but it becomes feasible to reconstruct the segment even if some of the packets are lost.

The sender REB node disseminates the packets as they are produced by the encoder and it also starts a timer that should expire in case retransmissions have to be performed, however this should only happen rarely for two reasons: (1) the erasure codes can recover from the loss of a subset of the transmitted packets; (2) a route monitoring mechanism is used to select those paths that expectedly have the highest quality of service (see Section 5.5.8). The receiver REB node accumulates the arriving packets in a receive queue, and when enough packets of a given segment are available, the receiver attempts to decode them. In case of success, it returns an acknowledgement back to the sender. Otherwise, it waits for the arrival of an extra packet for this segment before trying again to decode.

If the retransmission timer at the sender does indeed expire, then based on the latest information received from the acknowledgement, the sender node retransmits any unacknowledged packets (in this case the timer is reinitialized with a larger value). This process is repeated until the recovery of the original segment is accomplished. Depending on the network conditions, packets/segments may arrive and be decoded out of order. To address this issue, the REB utilizes a selective acknowledgement scheme to convey to the source information about which packets/segments have already arrived (see Section 5.5.6).

5.2.2 Ordered and duplication-free delivery

A *fair loss* network like the Internet does not ensure an ordered delivery of packets. This occurs because different packets, sent by one source, may experience distinct delays when transmitted through diverse routes. Additionally, there is the possibility of spurious transmission of duplicate packets, which often happens due to rerouting algorithms in intermediate nodes. Despite these difficulties, TCP provides FIFO ordering at the delivery and also removes duplicates. This is accomplished by assigning to each segment a sequence number, which is used on the receiver side to order the segments. Furthermore, the sequence numbers are used to detect and discard duplicate segments.

In the REB, the transmission of a segment corresponds to the dissemination of a fixed number of packets, which encode part of the original segment data and some redundant bytes (see Section 5.5.5). Each segment has an associated sequence number that is incremented monotonically. These sequence numbers are employed to keep track of lost segments and to detect data duplication (accidental or from replay attacks). Packets also have a distinct sequence number, which increases monotonically per segment. Therefore, a packet is univocally identified by carrying in the header a pair composed of the sequence number of the segment it belongs to plus its own sequence number (see Section 5.5.4 for more

details). Packets that arrive with the same identifiers are detected and removed as duplicates.

Within the same segment, packets that arrive out of order according to their sequence number are normally accepted by the decoding process, so unordered reception per segment is tolerated. FIFO ordering is enforced with the segment sequence number, and a receiver node can deliver data in the right order by buffering complete unordered segments until all their predecessors have been given to the application. The amount of unordered data that is maintained in a REB node is managed through a receiver sliding window flow control mechanism. This mechanism prevents the receiver from accumulating too much unordered data, something that could be used by a malicious REB node to overflow the memory of the receiver. The sender is informed of how much empty space is still available inside the window, and packets that arrive beyond this space are discarded. The flow control mechanism is addressed in detail on Section 5.5.7.

5.2.3 Authentication, data integrity and (optional) confidentiality

Unreliable networks cannot be trusted to guarantee security properties such as the authenticity, integrity or confidentiality of transmitted data from a source to a destination. An attacker may eavesdrop the communication to access sensitive data if she can intrude any intermediate node in the network path. Additionally, if the attacker has sufficient privileges, she can also alter transmitted packets or introduce new ones in the communication, in order to perform attacks such as intrusions at the endpoints or man-in-the-middle impersonations.

Secure protocols have been devised throughout the years to provide the necessary protection to communications over unreliable networks, for example some at the session level (such as SSL/TLS), and others at the network level (such as IPSec). Cryptographic mechanisms exist for guaranteeing the necessary security properties of data transmission, and these are usually employed at the communication endpoints but also at the intermediate nodes in some cases. The REB operates in a distributed environment where the nodes cooperate to increase the robustness of the communication by making use of multiple overlay paths to add spatial redundancy. Therefore, the REB needs to guarantee security properties on its own with the robustness aspect in mind.

At configuration time, a secret cryptographic key is assigned to every pair of REB nodes, which is shared exclusively by the two of them. This key is used to protect the communication from attacks, supporting the authentication of nodes and the integrity/authenticity of the data, and (optionally) its confidentiality. Every time a communication session between a pair of nodes is established, two cryptographic keys are created based on the shared secret key (plus some additional parameters, see Section 5.5.2).

One of the session keys is used on an optional basis to provide confidentiality of the data, where the content of a transmitted packet is encrypted at the sender and decrypted at the receiver, using the key. SIEM systems exchange mainly events and security information collected by the sensors. This data might be considered confidential in some organizations, while in others it can be of no concern. Since encrypting/decrypting every packet can introduce a reasonable performance penalty, and thus create delays that might be unacceptable, we decided to offer data confidentially on a optional basis. The application using the REB can indicate if it desires packets to be transmitted encrypted.

The other key is used to authenticate the nodes and provides the integrity/authenticity of the data. Every transmitted packet has a Message Authentication Code (MAC) appended, which is generated using the content of the packet and the key. MACs are verified at the receiver before packet delivery, and packets are discarded if their MACs do not match the expected values. Since transmitted data is divided into segments, which are subdivided into multiple packets (see Section 5.5.5), we could have

chosen to alternatively append a MAC to the whole segment and verify it only after the full reception. This solution would have the virtue of saving some MAC calculations, both at the sender and receiver, whenever segments are large. However, it suffers from a few drawbacks, making it less appealing in practice. First, the verification would be postponed, allowing corrupted packets to occupy space on the receiver buffers until much later. Second, the receiver node cannot separate good from bad packets, and therefore, a single damaged packet would cause the whole segment to be dropped (and then later retransmitted again).

REB nodes transmit packets using multiple concurrent routes, which can be based on a single direct channel between the source and the destination, or can have a channel from the source to an intermediary node and then another channel to the final receiver (see Section 5.5.3). Assuming that the sender and the receiver are both correct, then attacks can occur both on the network and on the intermediary node (in case this node was compromised). On direct channels, only one MAC is generated per packet by the source, using the authentication key shared with the destination node. On two-hop channels, a pair of MACs is created, the first for the intermediary node and the second for the destination node. After receiving a packet, the intermediary validates and removes its MAC and only then forwards the packet to the destination. Here, again, one could save a few MAC calculations if a single MAC was added independently of the type of route (i.e., direct or two-hop). However, since MACs are obtained relatively fast, we decided that it was better to have the capability to immediately identify and delete modified packets, both at the intermediary and final nodes. This feature is also helpful to determine if certain channels are under attack, since we can pinpoint with good precision where the fault occurred.

5.2.4 Robustness through multipath

The overlay network created among the REB nodes allows for multiple distinct routes (or paths) to be taken to transmit data to a specific destination. The source can, for instance, send the data directly or ask one of the other REB nodes to forward it to the destination. It is expected, in particular in large scale SIEM deployments, that these two paths will go through distinct physical links, and therefore, localized failures will only disrupt part of the communication. Based on this insight, the REB uses *multipath communication* to send data concurrently over several different routes in the overlay network (see Section 5.5.3). These routes consist either of direct paths between the source and destination nodes, or paths in which an intermediary node receives data from a source and redirects it to the destination. The REB resorts to a one-hop source routing scheme, in which the overlay route of each message is defined at the sender (source routing), based on the local knowledge of the state of the links, and is composed of at most one intermediate relaying REB node (one-hop). The option of having a single hop, i.e., a single intermediate node, is due to the conclusion of Gummadi et al. [56] that there is no considerable benefit in using more hops.

A REB node can be connected to the WAN through one physical link or through multiple links. The later case is often observed in organizations that operate in critical sectors, such as in electrical production and distribution, as a way to increase their resilience to accidental failures by ensuring that the links are physically separated (they contract the network service to multiple ISPs, and in some cases they go to the extent of using diverse network technologies, one wired and another wireless). If available, the REB also takes advantage of this form of redundancy by employing *multihoming communication*. Here, a node can either send or receive packets over any of the physical connections, thus increasing the number of possible paths that can be chosen when disseminating data to a certain destination. For example, if the sender and the receiver have two connections, then there are four alternative direct links between the

two nodes instead of a single one. It is expected that at least in part these links fail in an independent way, which means they can be used to provide correct packet delivery under somewhat adverse network conditions. Furthermore, multihoming at intermediary nodes is also exploited to increase the number of available overlay routes.

Given a certain segment m, the REB could send a copy of it over k distinct paths. This would allow $k-1$ path failures to be tolerated, at the cost of the transmission of $k-1$ extra segment replicas — the overhead is $(k-1) \times m$. Depending on the segment size, this cost can be significant especially when the network is behaving correctly (which is the expected normal case). Additionally, it also requires the receiver to process and discard $k-1$ duplicates, which can be a limitation in a SIEM system given the asymmetric data flow (remember that the core-MIS needs to receive all traffic coming from the edges). To address this difficulty, the REB employs *erasure coding techniques* to ensure that packet loss can be recovered at the receiver, at a much lower replication overhead (see Section 5.5.5). Basically, the sender needs to do some processing on the segment m to produce some extra repair information r, and then $m + r$ is divided into several packets that are transmitted over various links. Even if only a subset of the segment and repair data arrives, the receiver can still recover the original segment.

A sender REB node also periodically *probes* each of its most promising routes to a given destination to derive a quality metric to be associated with the path (see Section 5.5.8). Based on this metric, the sender can determine at each moment which are the best routes for a destination, and select them to disseminate a segment. Since the metric is calculated based on actual data collected from the network, it may require some period of time for the value of the metric to adjust after sudden changes in the network. During this period the sender could still think that a specific path is good, and consequently continue to use the path for transmissions, when in fact most packets could be lost. Moreover, a malicious intermediary REB node could attack the measurement process, for instance by making its paths look particularly good, and then suddenly start dropping all packets. In this scenario, the usage of erasure codes helps minimizing the impact of sudden changes in a route's quality, because if one assumes that the failure only affects a limited number of routes, the remaining ones still deliver enough packets for the original segment to be reconstructed.

5.3 REB interface

The REB is implemented as a Java library that can be linked with an application. Each REB node offers a relatively simple interface that contains the fundamental operations for transmitting and receiving data, as well as some auxiliary methods for the application to collect information about the system. Table 5.2 gives an overview of the main operations of the REB.

An initialization method from the library returns an active object after a REB node ID and a directory containing the necessary configuration files are provided (see Section 5.5.1). The ID represents a local node in the overlay and the returned active object acts as an agent for that local node, therefore the remaining methods are invoked on the object itself.

Furthermore, all methods can be safely called by multiple threads, which allows the application some flexibility for instance on how it handles communication with multiple remote nodes.

Initialization/Termination methods	
init(local_id, config_dir)	Initializes the REB object for the local node, on which the remaining methods are invoked. The parameter local_id indicates the identifier of a local REB node. The parameter config_dir indicates the directory containing the overlay network configuration files.
destroy()	Closes all communications and releases resources used by the REB object.
Communication methods	
send(buffer, size, destination)	Sends a message with size bytes from the data buffer to the specified destination node.
receive(buffer, size, source)	Receives a message with size bytes from the specified source node and stores it in the provided buffer.
probeIncomingData()	Blocks until some data is available to receive from some remote node, at which point the ID of the remote node is returned.
Information methods	
getLocalNodeID()	Returns the ID of the local node.
getLocalNodeAddresses()	Returns a list with all the IP socket addresses from the local node.
getRemoteNodesIDs()	Returns a list with the IDs of the remote nodes.
getRemoteNodeAddresses(id)	Returns a list with all the IP socket addresses from the remote node with the specified ID.
getRemoteNodeID(address)	Returns the ID of the remote node with the specified IP socket address.

Figure 5.2: Interface to the applications offered (per node) by the REB.

5.4 Sending and receiving data

The procedure of sending/receiving messages in the REB encompasses a number of steps in order to enforce the various properties discussed in the previous sections with a simple to use application level interface. Since REB nodes are organized as an overlay network, the sender typically has several routes available for data transmission, which in most cases are expected to correspond to disjoint physical links. It is anticipated that most of these routes will fail independently with a reasonable probability (see Section 5.5.3). By forwarding data over a subset of the routes concurrently, the REB is able to tolerate some failures in a transparent way, and at the same time support the distribution of the network load over the alternative paths. Routes are selected using a metric that takes into consideration the observed RTT and loss rate in the recent past (see Section 5.5.8). The REB also uses erasure coding to create multiple blocks of data from the messages, which have in total a slightly larger size than the original message, and that are transmitted individually through the chosen routes (see Section 5.5.5).

5.4.1 Data transmission

The transmission process, which implements how individual messages are delivered from a source to a destination, is illustrated in more detail in Figure 5.3. When an application needs to send a message, it calls the REB interface and indicates the destination node and provides a buffer with the data, which is read as a stream of bytes (see Section 5.3). Multiple calls can occur concurrently if the application has

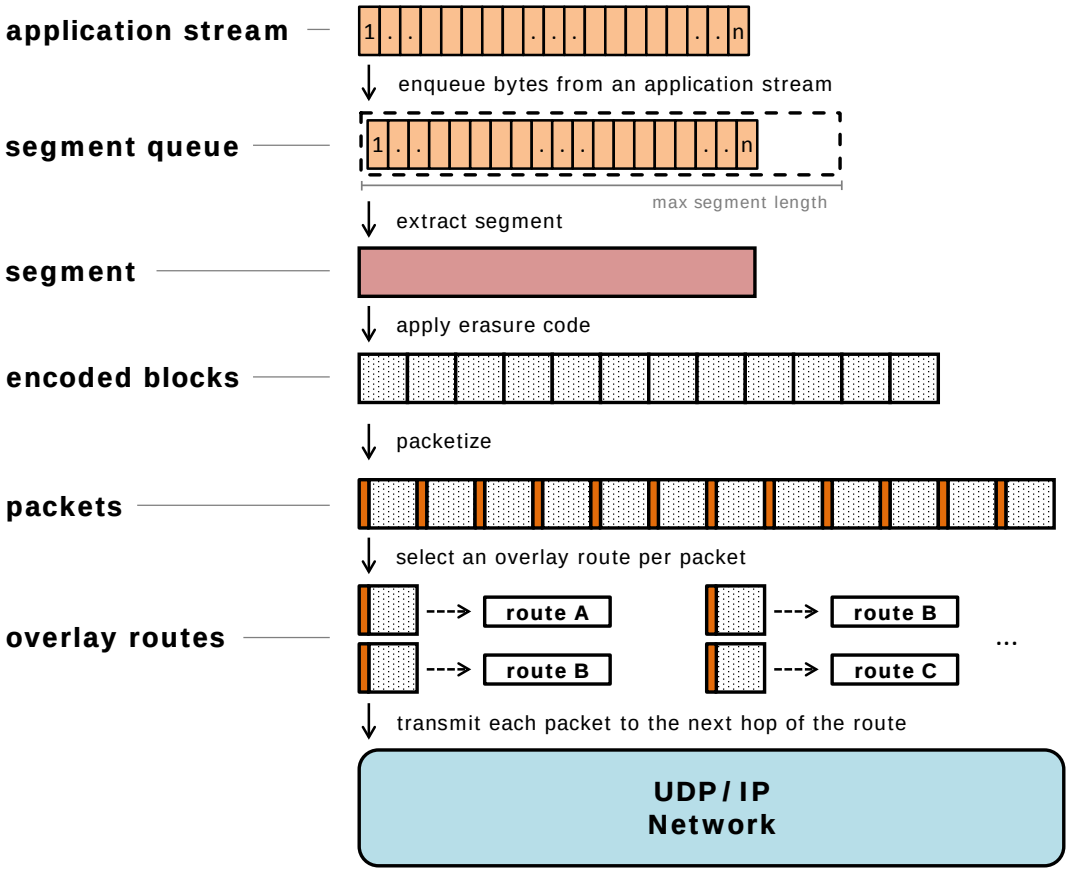

Figure 5.3: The data transmission process at a REB node.

more than one thread. When the call returns, the application can reuse the buffer because either it has been transmitted or its contents have been locally copied to a send queue (named *segment queue*).

The REB maintains multiple segment queues, one per each active destination, and their function is to accumulate application stream bytes (in FIFO order) until it is possible to create a segment with an optimal size for transmission. After the bytes are copied into a segment queue, a flow control mechanism dictates when a segment should be created from the available data (see Section 5.5.7). As an optimization, the REB may skip this copy and transmit the bytes immediately, if it is possible to create a segment directly from the application stream and the flow control restrictions allow it to be transmitted.

A segment queue is mainly used in two situations. First, in case the network is occupied with the transmission of older messages, a local copy is created so that the send operation can return, allowing the application to continue to run. However, if a message is too large and does not fit entirely inside the queue, it is split so that part of it fills the queue and the rest is scheduled to be inserted when there is space again. Second, if the application normally sends small messages, the queue is utilized to accumulate more bytes. Ideally, one would like to increase efficiency by transmitting packets with a size approximately equal to the MTU of the underlying network, which has the advantage of reducing the per-packet load experienced by intermediary routers. Additionally, the coding algorithms require a reasonable amount of data for optimal execution. This amount of data is named the *maximum segment length* and its value has to take into consideration the size of the headers, the type of code and the network MTU between the

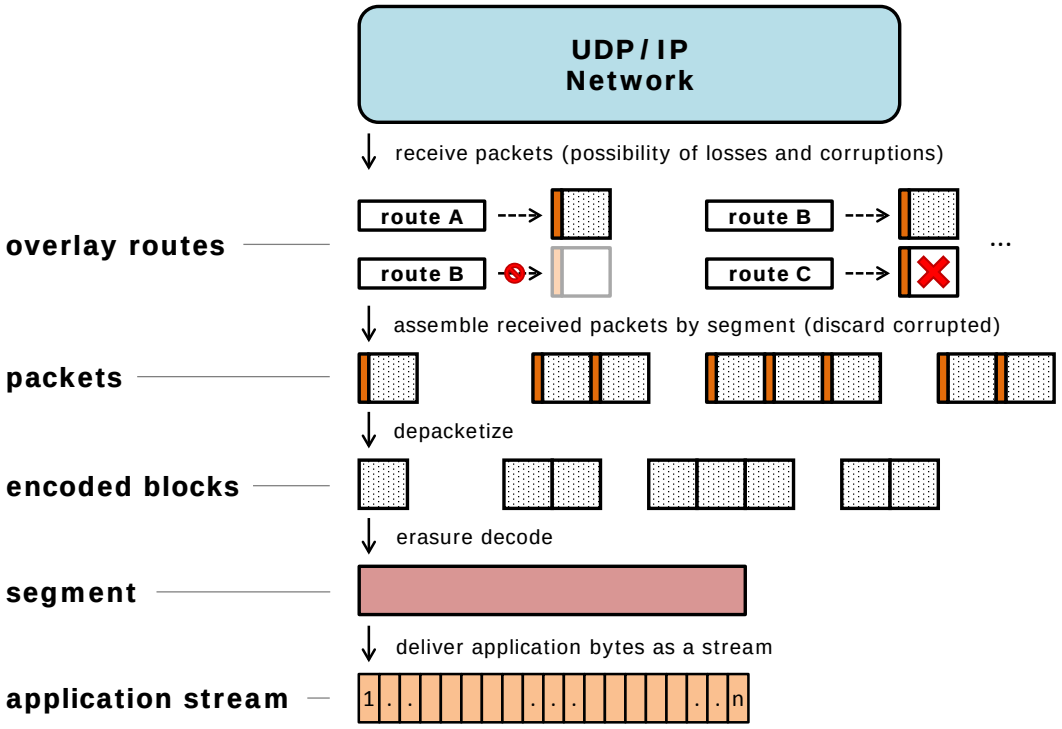

Figure 5.4: Scheme with the data receiving process.

sender and receiver.

The extracted segment is then encoded by applying an erasure coding algorithm. The algorithm divides the segment in multiple blocks, and then processes them to produce the encoded blocks. In the REB, we use a systematic version of erasure coding, which means that the first encoded blocks correspond exactly to the original segment (see Section 5.5.5 for more details).

Encoded blocks are then packetized by adding a header that contains, among other fields, the identification of the packet (see Section 5.5.4) and the final receiver. In order to maximize the reconstruction capability of the erasure code, each encoded block should be placed in a distinct packet. This ensures that a packet drop in the network only affects one block, allowing the remaining ones to recover the lost data. In some cases, for efficiency reasons, it makes sense to include more than one encoded block per packet. This is only valid if two (or more) blocks fit inside the MTU of the network. This optimization is utilized however with some care because it can weaken the failure independence assumption on which erasure codes are based.

If the segment is very small, it might be difficult to apply the most sophisticated coding algorithms, since they may be optimized for larger data sizes. In this case, it is more efficient to place the whole segment inside a packet, which is then replicated enough times to tolerate a certain number of failures.

The sender next looks at the available routes to the destination, and selects a limited number of the best routes, allowing the segment to arrive as fast as possible. A MAC is added to the header to let the final receiver detect integrity violations. In case an intermediary node needs to forward the packet, then a second MAC is also appended. The packet is then transmitted over the corresponding overlay route, which translates in a transmission through the underlying network using UDP over IP.

5.4.2 Data reception

On the receiver side, the node collects the packets arriving from the various routes as represented in Figure 5.4. The node is prepared to receive only a subset of the packets, since some of them may be lost in the network, while others can be corrupted. This second problem is detected with the included MAC, and the corresponding packet is deleted. The MAC is also employed for the protection of attacks where an external adversary (not controlling a REB node) may generate malicious packets, for instance, to confuse the segment reconstruction procedure or to make a DoS. Since the adversary does not share a key with the REB node, she is incapable of producing packets with the right MAC.

DoS attacks performed by a malicious REB node are addressed with a flow control mechanism (see Section 5.5.7 for details). When requested or piggybacked in the acknowledgements, the receiver indicates to the sender the amount of bytes that can be transmitted and that fall within the associated receive queue. Packets arriving when the queue is full are simply discarded. Additionally, packets with identifiers outside the expected range are also dropped, therefore, averting attacks that try to exhaust the memory by extending the local queues.

Packets can be received for the local node or to be forwarded to some other destination. In the first case, a few correctness checks are carried out, including duplication removal, and then the header is removed to obtain the encoded block. Next, the block is stored in the receive queue associated with the sender. Alternatively, when the node acts as a two-hop router, the packet is also checked and then enqueued to be transmitted to the final destination. To thwart starvation attacks caused by a malicious sender node, where it could try to delay the transmissions of this node, the packet is put at the end of the queue to wait for its turn.

Each encoded block is assembled according to the particular segment that it belongs to. Since we are using erasure codes, we try to decode as soon as enough blocks have arrived. However, sometimes, it may happen that it is impossible to reconstruct the segment with the available blocks. When this happens, the receiver needs to wait for the arrival of more blocks (or for the retransmission of lost blocks), and then perform the decoding operation again. As segments are decoded, they are stored in the receive queue waiting for the application to read them as a stream of bytes.

5.5 Communication mechanisms

This section describes the details about the mechanisms utilized by the REB. It explains how the network is configured and how communication sessions between pairs of nodes are established. Details are provided about the usage of erasure codes, as well as how multipath contributes to the overall robustness of the communication. It also indicates how data packets are identified and how acknowledgments are handled, and explains the flow control mechanism. Lastly, it presents a mechanism for route probing, supporting the inference of a quality metric for individual routes, which is then used by a route selection algorithm.

5.5.1 Overlay network configuration and setup

The REB uses a static configuration for the overlay that defines the set of nodes that may participate in the communications (some of them may be down or disconnected). Each node is identified by a numerical

ID, which is used by the application to refer to a specific node in the REB interface (see Table 5.2). These numerical IDs are chosen for instance by a network administrator and placed in a unique configuration file that is replicated on every REB node machine (named the *node file*). Additionally, every pair of nodes shares a secret cryptographic key for secure communication. These keys are also chosen for instance by a network administrator (possibly randomly generated) and placed in a separate configuration file per REB node (named a *key file*).

Both the *node file* and a *key file* are read by a REB node at the moment of its initialization, making the node ready for communication with other configured nodes afterwards. Details about both types of file follow:

Node file A REB node sends/receives packets through a specific network address, comprising an IP address and UDP port. The ports can be different across the overlay, depending on the machine where the node is located. If a machine has multihoming, then several IP addresses may be assigned, one for each physical connection. The *node file* is used for assigning one or more network addresses to a REB node ID.

Key file Each of these files has the numerical ID of the local node in its name and contains a list of keys, each mapped to a specific remote node ID. To facilitate the configuration of these keys, we provide a tool that receives a list of node IDs as input and automatically creates unique keys for each pair of nodes, generating a *key file* per node. With the help of this tool, a network administrator needs only to transfer each generated file to a secure location at the respective node's machine.

5.5.2 Session management

At start-up, a REB node is connected to no one, and a communication session is only established between two REB nodes when the source initiates a transfer of data to the destination. A session protects the communication through the use of two cryptographic keys, one that provides node authentication and message integrity (the authentication key), and another that optionally provides confidentiality (the encryption key). Establishing a new session consists in resetting any previous state and in setting up the two keys. The former is necessary because nodes are allowed to crash and then later reinstated in the overlay. It could occur that the destination node had been exchanging packets with the source, and then the source had to reboot. In this case, during the new session establishment (initiated by the source), the destination node would become aware of the problem, and therefore it would clean information maintained on behalf of the source[1].

A session only protects the communication in one direction (from source to destination), which includes the data transmitted by the source, as well as acknowledgments transmitted back by the destination. If two nodes wish to transmit data to each other, then they must establish two different sessions. For this reason, it is possible for a logical connection between two nodes to be in a "half-open" state, in which only one side is transmitting data. We differentiate sessions based on the direction of data flow because sessions may need to be refreshed more or less often according to the volume of data being transmitted by a node. Therefore, a session that is refreshed more often than the other in the opposite direction (due to a larger flow of data) does not affect the behavior of the latter.

[1]The cleaning includes discarding pending packets and out of order segments, and other management data (see Section 5.5.7). Completely reconstructed segments that are stored in order in the receive buffer cannot be deleted because the sender might have the expectation that they will eventually be delivered to the application.

Session handshake To set up the two session keys (for authentication and encryption), a source node initiates a handshake protocol, exchanging with the destination certain parameters that are used in the creation of symmetric keys at both sides. The parameters include the IDs of both nodes, as well as nonces. The IDs differentiate session keys from different pairs of nodes, and nonces differentiate keys — within the same pair of nodes — that are created at different times. Nonces are obtained by concatenating two values, a locally generated secure random number and a high resolution timestamp (up to the microsecond).

Figure 5.5 illustrates an exchange of handshake messages between a source and a destination node. The protocol begins with the source node generating a nonce $N1$, and then sending to the destination a special message $INIT$ containing that nonce and the IDs of both nodes. The destination node responds by generating another nonce $N2$, and sending back to the source a special message $INIT\_RESP$ containing both nonces and both IDs. When the source node receives this message, it authenticates the destination because the received nonce $N1$ matches its local one. The source node then responds by sending to the destination a special message $RESP$ containing both nonces and both IDs. Similarly, when the destination node receives this message, it authenticates the source because the received nonce $N2$ matches its local one, and the received nonce $N1$ matches the one received previously in the $INIT$ message. All the exchanged handshake messages are authenticated with MACs obtained from the unique secret key K_s that is shared by both nodes at configuration time. At the end of the handshake protocol, both nodes agree on their IDs and values for the nonces $N1$ and $N2$, and establish the two session keys in the following way (strings "A" and "E" are used to differentiate the two keys):

$$K_{Auth} = Hash(K_s, N1, N2, ID_{Src}, ID_{Dest}, \text{“}A\text{”})$$
$$K_{Encr} = Hash(K_s, N1, N2, ID_{Src}, ID_{Dest}, \text{“}E\text{”})$$

The new keys substitute the old ones when the handshake finishes. Packets from the previous session may still be in transit when this occurs. These packets will be discarded upon arrival because the MACs are no longer valid. This mechanism has the benefit that prevents packets from an older session from confusing the receiving algorithms.

Extra parameters Some handshake messages may carry extra parameters that convey additional information to the receiver node. These parameters are not used in the creation of new session keys, but instead ensure that both source and destination nodes agree on some value before any data is delivered within the newly established session.

One of such parameters is the receive window value of the receive queue from the destination node. This value is transmitted to the source node inside a $INIT\_RESP$ message, and is used by the source as the initial remote receive window for data communication (see Section 5.5.7 for more details).

Another parameter is a flag indicating whether the established session is a refresh of the previous one, or is a new one. A session refresh is initiated by the source node if the segment sequence number reached its maximum value for the current session (see Section 5.5.4). This flag is transmitted to the destination node inside a $RESP$ message, and changes the behaviour of the destination upon reception of this message. If the flag is inactive, then the destination resets the current session that was previously initiated by the source (if any), discards any incomplete or unordered received data and notifies the application of the fact. If the flag is active, then the destination resets the previous session, but does not discard any data or notifies the application. In this case, it is required that no incomplete or unordered received data exists at the destination, which is ensured if the source node behaves correctly and only initiates the session refresh after all outstanding data is acknowledged (see Section 5.5.6).

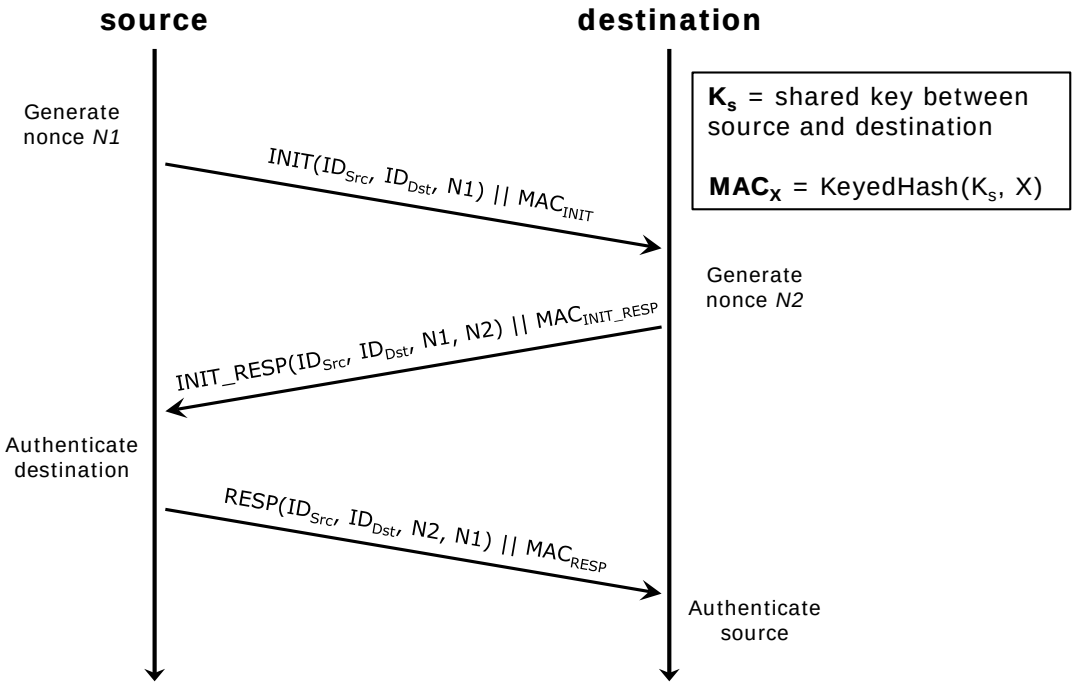

Figure 5.5: Handshake protocol for a new communication session.

Security considerations It may happen that one of the handshake messages is lost in the network. To address this problem, the sender of the message is responsible for its retransmission until the other side responds. So, for instance, if message $INIT$ is dropped, the source node should periodically resend it until the following handshake message arrives. If the destination node sees a duplicate $INIT$, this could either indicate that $INIT\_RESP$ was lost or delayed. In this case, it simply waits for its retransmission timer to expire, which would cause $INIT\_RESP$ to be resent, or for the arrival of $RESP$ that would conclude the handshake.

The loss of the last message, $RESP$, is recovered in a slightly different manner. From the point of view of the destination node, it cannot distinguish the case when $INIT\_RESP$ or $RESP$ are dropped by the network. Therefore, it keeps retransmitting $INIT\_RESP$ until a $RESP$ arrives. At the source node, when a duplicate $INIT\_RESP$ is received, the $RESP$ message is resent. In all situations, handshake messages are only retransmitted a certain pre-defined number of times. When the limit is reached, the connection attempt is aborted and the application is notified of the fact.

An adversary could take advantage of the handshake protocol to attack the REB communications. For example, she could replay an old $INIT$ message to fake a reboot of the source node to force a session reset. Since in general the destination node cannot distinguish a replay from a valid connection attempt, it starts the handshake protocol by responding with the $INIT\_RESP$. However, it continues to process the packets arriving from the source as usual, ignoring for now the $INIT$. Since the adversary does not know the shared key, she cannot produce a valid $RESP$. Therefore, after a number of $INIT\_RESP$ retries, the destination node forgets about the connection.

In another example attack, the source node could send a $INIT$ message, and then the adversary could replay an old $INIT$ of the node. As a consequence, the destination would receive two valid but different $INIT$ messages from the same node. In this case, we use a simple arbitration procedure, where

the handshake corresponding to the $INIT$ carrying the nonce with the largest timestamp is the one that is executed, and the other is disregarded[2].

A node simply ignores a message for which the corresponding previous handshake message was not received (if a $INIT\_RESP$ or a $RESP$ arrives without having sent the $INIT$ or $INIT\_RESP$, respectively). Additionally, since a node only resends a message a predefined number of times, this prevents denial of service attacks where an adversary keeps replaying the most recent handshake messages (either $INIT$ or $INIT\_RESP$) to force the destination to perform retransmissions.

5.5.3 Multiple paths and multihoming

A source REB node uses one or more direct paths from the underlying UDP/IP network to transmit data to a destination node, but also relays the data to other intermediary nodes — which forward it to the destination — in order to create additional routes in the overlay network, thus increasing the robustness of the communication. If the REB is able to determine which routes are behaving erroneously, and picks alternative routes for data transmissions, it is possible to tolerate failures in the network. Of course, these measures are only effective if the failures do not completely cut all communications.

The REB resorts to a one-hop source routing scheme, in which an overlay route is defined at the sender, based on the local knowledge of the state of the links, and is composed of at most one intermediate relaying REB node. Therefore, the available paths are the following: first, there is the direct link from the sender to the receiver; second, since any other node can act as an intermediate router, each one of them defines an extra path. Overall, in a REB deployment with n nodes, there are at least $n - 1$ paths connecting every pair of nodes.

REB nodes may also be interconnected via multihoming, i.e., by two or more distinct physical links (e.g., a REB node could have a pair of network interfaces and would be connected through two different ISPs). Since these links usually only share a minimum amount of resources, their failure can be considered independent in many scenarios (e.g., a DoS is performed in one of the ISPs). The REB takes advantage of multihoming to increase the number of available overlay paths, allowing a node to overcome the failure of one of its links by exploring alternative routes.

The number of network interfaces directly influences the quantity of overlay paths a local node may have at its disposal. Figure 5.6 shows an example overlay network configuration that makes use of multihoming in order to provide a diversity of links. REB nodes with identifiers ID1 and ID3 have a single network interface, while nodes ID2 and ID4 have two interfaces. Therefore, node ID2 can reach node ID1 through two direct links, and can send packets to node ID4 over four direct paths.

The exact number of overlay routes that exist can be calculated in the following way. Let y be the number of interfaces on a certain local node and w the number of interfaces on another remote node; in total, there are $y \times w$ available direct paths between the nodes. Two-hop paths include an extra intermediary node and make use of the same interfaces as in the direct paths, as well as the interfaces of the intermediate node (the packet can arrive in one interface and then leave from any of the available network interfaces). Let z_i be the number of interfaces on a certain intermediary node. Then, the number of paths that can go through this intermediate are $y \times w \times z_i^2$. In total, the number of available paths between the two nodes are:

[2]Here, we are working under the assumption that the adversary cannot change the clock of the source node to a later time, and then collect a $INIT$ message with a larger timestamp. If this assumption is violated, then a new shared key needs to be setup between the two nodes by the SIEM administrator. Notice, however, that even in this case the adversary cannot complete the handshake protocol and deceive the two parties.

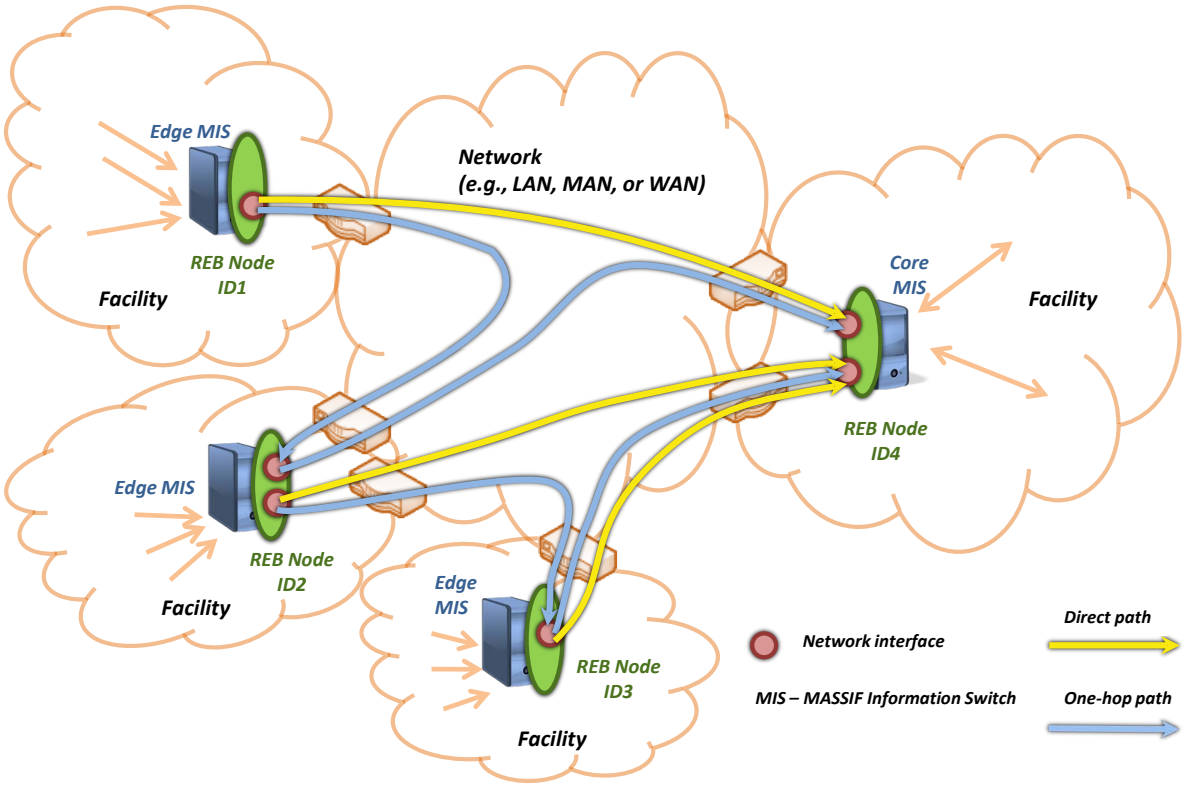

Figure 5.6: Multihoming in REB.

$$Total\ paths = y \times w + \sum_{i=1}^{n-2} y \times w \times z_i^2$$

Returning to the example from the figure, the total number of paths between nodes ID1 and ID4 is equal to 12. There are two direct paths; eight two-hop paths through node ID2; and two two-hop paths through node ID3. For overlays with many nodes, the growth in number of paths could become a challenge if all of them were used in the communications. However, only a subset of the paths is actually employed by a node to transmit data, and these are picked based on their quality metric (see Section 5.5.8).

5.5.4 Identification of segments, blocks and packets

In the REB, a sender node receives data as a stream of bytes from the application and transmits those bytes in packets over the network. A receiver node must therefore be able to reconstruct the original bytes upon reception of individual packets, so that it can deliver the data in the form of a stream back to the application. Since the REB applies erasure coding to the data before transmitting it in packets, and since erasure codes are optimized for large data sizes and require a fixed number k of data *blocks* (see Section 5.5.2), we must aggregate as much data as possible from the application stream before passing it to the erasure codes. We name these data aggregations as *segments*.

When a segment is passed to the erasure codes, two things happen: (1) the segment is divided into k source blocks with equal size; (2) the source blocks are encoded into an arbitrary number of encoded blocks with the same size as the former ones. These encoded blocks are transmitted inside individual packets and trigger a decoding procedure at receivers in order to obtain the original source blocks.

The size of a segment should be as large as possible, such that an encoded block completely fills the payload of a packet. The payload capacity of a packet is configured as the Maximum Transmission Unit (MTU) of the underlying network minus the size of the IP, UDP and REB headers. In other words, let B_{len} be the length in bytes of a (source or encoded) block with the largest allowed size, less than the MTU. The maximum size of a data segment is then: $k \times B_{len}$ bytes.

To summarize, application data is aggregated into segments, then split into source blocks that are used to generate encoded blocks of the same size, which are carried inside individual packets. Segments and blocks must therefore be univocally identifiable and capable of being ordered, in case duplicates reach the receiver or data is received out of order. Consequently, segments and blocks are identified using sequence numbers, and receivers are informed of these numbers by reading them in the information conveyed inside packet headers.

Segments The REB assigns to each data segment a sequence number that is represented as a 24-bit unsigned integer, starting with the value 1 and being monotonically incremented per segment until a maximum value of $2^{24} - 1$ (the sequence number 0 represents a null segment).

Segments are obtained from the stream of bytes provided by the application, aggregating as much data as possible each. If the application is continuously sending bytes at a constant rate, then segments are expected to be maximally sized, which is in the order of few MB. Under these circumstances, the segment sequence number should reach its maximum value only after at least a few TB of data are transmitted. Nevertheless, a sender REB node is prepared for such an event and refreshes the communication session with the receiver when that happens, resetting also the segment sequence number to its initial value for the new session. Sequence numbers may be safely reused, without the danger of introducing corruption of data caused by replay attacks, because packets carrying sequence numbers from the previous session will not be accepted as the MAC is invalid.

Additionally, the choice for the space size of the segment sequence number took into account the information provided by the MASSIF use case scenarios [25]. For example, the Olympic Games scenario, at its pick load, produces around 12 million events per day before aggregation. If these events were to be transmitted in separate segments, something highly unlikely because there is typically aggregation of events at the collectors, then communication sessions would only have to be refreshed with a frequency less than once a day.

Blocks Each encoded block, obtained from the source blocks of a specific segment, is assigned a sequence number that is represented as a 16-bit unsigned integer, starting with the value 1 and being monotonically incremented per packet until a maximum value of $2^{16} - 1$.

The erasure codes require that source blocks be identifiable and capable of being ordered upon decoding so that the original segment may be correctly reconstructed. The codes ensure this by assigning a position (or index) within the segment to each source block, ranging from 0 to $k - 1$. Since the erasure codes we use are systematic, the first k produced encoded blocks correspond exactly to the original k source blocks. The sequence numbers for those blocks are therefore aligned with the indexes of the source blocks (although offset by one), and so we simply use them instead of specifying indexes inside the codes.

The REB does not expect that block sequence numbers overflow, because the erasure codes limit the number of encoded blocks generated from a segment and that limit does not surpass $2^{16} - 1$.

Packets A data packet typically carries an encoded block belonging to a specific segment. Sometimes, a packet may carry multiple encoded blocks if their size permit it. More rarely, a packet may carry an entire segment if its size is small enough to fit inside the payload. In any case, a data packet always carries in a header a segment sequence number and (if applied) the sequence number(s) of any encoded block(s) inside it. This ensures that data is properly identifiable and capable of being ordered by the receivers.

5.5.5 Erasure codes

A REB node sends packets concurrently over a few of the available routes to the destination. However, data transmission through multiple routes per se is not sufficient to achieve robustness in the communications. In fact, even a single route behaving erroneously (e.g., losing packets) is enough to prevent the original data from being reconstructed. Therefore, using multiple concurrent routes can actually degrade the reliability of the whole communication. Two possible approaches to recover from losses are (1) the retransmission of the packets at a later time, or (2) the concurrent transmission of several copies of the packets over different paths. The first solution has the advantage of minimizing the amount of data that is sent, at the cost of delaying the delivery of the packets (since retransmissions occur after a timeout). The second approach has the opposite characteristics.

Using Forward Error Correction (FEC) codes allows for the detection and recovery of bit/packet-level errors during transmission at the expense of sending extra redundant bits/packets and extra computational power. By carefully choosing an FEC code, it is possible to reduce the amount of redundancy while tolerating a certain amount of data loss. With well devised FEC algorithms, it is also possible to minimize the extra time spent during detection and recovery phases at the receivers, allowing that extra time to be far less than the average transmission latency time.

The REB uses a special type of FEC codes, the *erasure codes*, to both decrease the amount of transmitted data and minimize the delays in case of losses. Erasure codes work by splitting data into a number k of source blocks and then producing, from those, a number N of encoded blocks (with the same size as the former) that are transmitted over the network. Recovery of the data is possible by receiving only a subset K of the N encoded blocks. Generally, $k \leq K \leq N$.

In order to operate correctly, erasure codes must assume an underlying *binary erasure channel* (BEC), in which data elements (in this case packets) are either received without errors, or not received at all. Unfortunately, this property cannot be ensured by the underlying UDP/IP network alone, since UDP only provides (sometimes even optionally) a checksum field inside a datagram header, which is not sufficiently robust or secure to ensure the integrity of the payload data (e.g. an adversary could modify a datagram in transit and replace the checksum value with a new calculated one). However, the REB transmits packets with appended MACs to provide, for instance, the integrity of received data, and so a receiver is capable of detecting — in a robust and secure way — errors or modifications in received packets, discarding those that fail the integrity checks. A BEC can thus represent this scenario, which makes the usage of erasure codes ideal.

Fountain codes The reception efficiency of an erasure code is given by the number K of encoded blocks required by a receiver in order to successfully recover the original k source blocks. As K ap-

proaches k, the reception efficiency increases and reaches an optimal value when $K = k$. An optimal erasure code holds an optimal reception efficiency, where it is sufficient for a receiver to collect any k of the transmitted encoded blocks to recover the k source blocks.

Optimal codes, however, manifest sub-optimal memory- and time-complexities, which hinder communications when transmitting large sized segments of data. For this reason, we chose to use *near-optimal erasure codes* instead, that sacrifice reception efficiency for better memory and time constraints. Using such codes, receivers can attain the successful recovery of k source blocks by receiving $(1+\epsilon) \times k$ encoded blocks, where $\epsilon \geq 0$. In practice, though, the value of ϵ can be made small (e.g. $\epsilon = 0.05$) while granting a very high probability of successful recovery.

A class of near-optimal erasure codes is the *Fountain Codes* [71], which have the additional property of being rateless. The code rate R of an erasure code corresponds to the proportion of the total transmitted data that is useful (non-redundant), and is given by $R = k/N$, where k is the number of source blocks (corresponding to the useful data) and N is the number of transmitted encoded blocks. Erasure codes such as Reed-Solomon codes [66] require a fixed code rate, which must be defined a priori according to a predicted loss rate of the communication channel. Rateless codes, on the other hand, do not fix the code rate, which allows them to adjust to an increase of the data loss rate by simply transmitting more encoded blocks (i.e. by increasing N). This works well in the presence of multiple communication channels that exhibit independent loss rates, specifically in content delivery systems, where one sender transmits some content to many receivers. In such systems, the rateless property is crucial for a scalable configuration because it simplifies the sending process and receivers do not need to query the sender for specific pieces of the content [71].

The REB can also take advantage of Fountain Codes because a sender node uses multiple routes to transmit data, and these routes have typically independent loss rates. However, a sender node transmits multiple segments of data sequentially (obtained from the application stream) to a receiver, which means that the number of distinct encoded blocks per segment must be bounded in order to make progress. Furthermore, REB nodes may be compromised and might try to crash other nodes by transmitting excessive amounts of data and exhausting the available memory at the receivers, so to prevent that from happening, the REB enforces a flow control mechanism (see Section 5.5.7) that also requires an upper bound on the number of transmitted encoded blocks. This upper bound N_{max} is configured at the REB to be sufficient for a very high probability of successful recovery by receivers. In practice, though, correct REB nodes will send less than N_{max} encoded blocks on average, while adjusting the number N of transmitted blocks according to the perceived loss rate of the routes (see Section 5.5.8).

Erasure codes may also be systematic, meaning that the set of transmitted encoded blocks includes the set of source blocks. This has the advantage of making the reception perform faster (when the communication channel does not incur loss of data) since it avoids the recovery process at receivers altogether. The REB implements a systematic version of the fountain codes.

Implementation details The REB resorts to a particular implementation of Fountain Codes: a systematic version of the *LT codes* [70]. In order to use the LT codes, the REB splits each data segment into k source blocks, which are then encoded into N encoded blocks that are transmitted over multiple routes. Since we use a systematic version of the code, the first k encoded blocks are the source blocks, and are referred to as *systematic blocks*.

Since it is possible for a data segment to have a size that is not a multiple of k, it is necessary to add some padding bytes to the segment before splitting it into source blocks. The size of the padding is at most $k - 1$ bytes, but is not explicitly transmitted by a sender node, since the receiver is informed of the original segment size and can thus automatically account for any padding. An exception to this rule

is when a block contains both data and padding. This process is, however, transparent to the encoding and decoding procedures, since source blocks that contain only padding are treated as usual for decoding purposes, regardless of being transmitted or automatically inferred by receivers.

At a sender node, if sufficient data is available in the segment queue, then the segment size is selected in such a way that every transmitted packet (containing the encoded block plus some additional information) has a size near to the Maximum Transmission Unit (MTU) minus the size of the headers (added by the REB, UDP and IP). This ensures an efficient utilization of the network, and also that a packet drop only affects a single encoded block, something that is typically assumed by the codes.

LT codes are, however, able to recover from bursts of encoded block losses. Therefore, if the segment is small, one can put a few encoded blocks in the same packet to reach a dimension similar to the MTU (note that the random functions used by the LT codes are devised for particular values of k, and therefore it is not possible to simply decrease k without affecting the properties of the code). This way one can keep the network efficiency, at the cost of potentially losing more encoded blocks. For tiny sized segments that fit in a single MTU, there is no advantage of employing coding algorithms. In this case, we simply replicate the segment over a few packets and send them concurrently through distinct routes.

Encoding procedure Each source block is identified by a number between 1 and k, and each encoded block is identified by a number between 1 and N_{max}. The systematic blocks are encoded blocks with numbers between 1 and k.

A (non-systematic) encoded block is created by calculating the XOR of some of the source blocks. Two random functions are called, one to determine the number of source blocks that should be XORed (the degree of the encoded block), and the other to select the actual source blocks. The functions ensure that — with a probability that increases as the number of encoded blocks grows — the set of all source blocks is contained in the set of encoded blocks, and there are enough encoded blocks of degree 1 (these are crucial for the success of the decoding procedure).

When a sender node transmits a (non-systematic) encoded block, it needs also to send information about which source blocks it encodes. In order to reduce the size of this information and to keep it constant regardless of how many source blocks are encoded, the sender appends a pseudo-random seed value to the encoded block data before transmitting the block. This value is used to initialize the pseudo-random number generator (PRNG) used by the random functions during the encoding procedure, and is also used by the receiver node to initialize an equivalent PRNG in order to obtain the same information as the sender about the source blocks.

Decoding procedure Decoding follows a *belief-propagation algorithm* and is performed gradually, as the encoded blocks arrive at the receiver. In each step, it is checked if the new encoded block allows the recovery of a source block, and if this is the case, this knowledge is further propagated to decode other blocks. Similar to the encoding procedure, decoding is done by calculating the XOR of received encoded blocks with already decoded source blocks. Eventually, when enough encoded blocks reach the receiver, it is possible to reconstruct the original data and an acknowledgment is sent back to the sender node. If decoding is unsuccessful, then the sender is responsible for retransmitting some of the missing encoded blocks (see Section 5.5.6).

5.5.6 Acknowledgments and Retransmissions

The data provided by a sender application to the REB is split into segments and delivered sequentially, segment by segment, to a receiver application. More specifically, a sender REB node first splits the input data into segments, then splits each segment into k source blocks, applies erasure coding over the blocks to produce N encoded blocks, and finally transmits each encoded block inside a packet over the network. A receiver REB node receives the individual packets from the network and reconstructs the original segment from the received encoded blocks. Until such reconstruction is complete, a receiver node stores the received data in memory, and when the full segment is ready, it sends an acknowledgment back to the sender and delivers the segment to the application.

The use of erasure coding and multiple overlay routes offers a good level of robustness against lost packets. Ideally, the combined loss rate of the used routes is bounded, such that an informed number N of transmitted encoded blocks by a sender node is enough to overcome packet loss, allowing an immediate reconstruction of the original segment at the receiver. However, the routes may suddenly behave worse than expected, and may end up discarding too many packets, preventing the segment from being rebuilt at the receiver. To overcome this problem, the sender node needs to retransmit some blocks if it does not receive an acknowledgment within a predetermined time.

Receive queue The underlying UDP/IP network may reorder the packets being disseminated. Additionally, the use of multiple routes in the overlay may also contribute to the arrival of out of order packets, as the paths can have diverse transmission times. When using erasure codes to reconstruct a segment, a receiver node tolerates unordered reception of packets in a natural way, however the unordered reconstruction of individual *segments* must be handled specifically. Therefore, it is advisable that receivers keep unordered data segments (fully reconstructed or in the process thereof) in buffers instead of dropping them. In the REB, a separate buffer is managed for each source at the receiver, and is named *receive queue*.

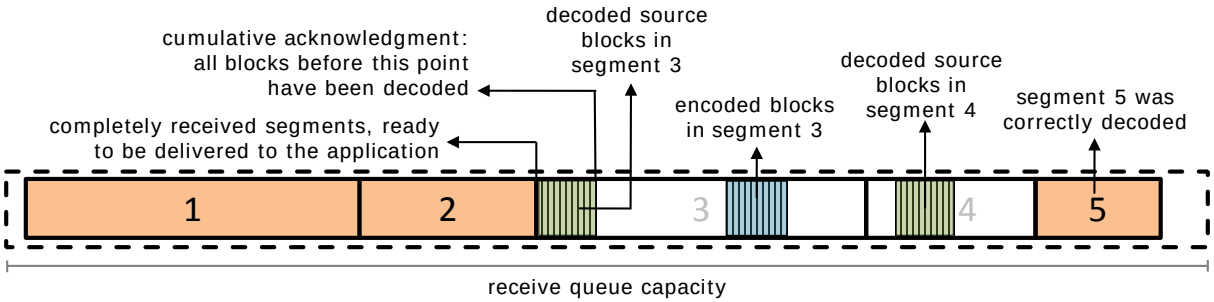

Figure 5.7: Example of a receive queue.

The receive queue has a fixed length in number of bytes, and allows for the storage of up to a given number of (maximum sized) segments. The queue may contain segments completely decoded or segments that have been partially received (see an example in Figure 5.7). As the application reads the decoded data (in FIFO order), the corresponding space is freed so that further segments can be accommodated. The segments are placed in the queue according to their sequence numbers (see Section 5.5.4), and consequently there might be holes in the queue so that out of order segments can be added at a later time. Packets from segments that would need to be stored beyond the queue limit are simply dropped as

the queue is not extended to save them[3].

The decoding process applied to received data transforms encoded blocks into source blocks that allow for the reconstruction of the original segment at the receiver (see Section 5.5.5). Eventually, when k source blocks are decoded, the whole segment is complete and ready to be delivered to the application (as long as there are no other incomplete segments ahead in the receive queue). Both source and encoded blocks are placed inside the respective segment, occupying a certain amount of bytes each in the receive queue. Source blocks are placed at the appropriate location, according to their index ranging from 0 to $k-1$. Encoded blocks have no particular location inside the segment so they are simply placed in the order of arrival after the position of the source block with index $k-1$ (in here the ordering of the encoded blocks is irrelevant because they are not delivered to the application).

Selective acknowledgment The REB informs the sender about which blocks/segments have been correctly received with a *selective acknowledgment (SACK)* (a mechanism somewhat inspired from TCP [74]). A SACK contains, for each partially received segment (identified by its sequence number), a list of indexes (ranging from 0 to $k-1$) defining specific source blocks within the segment that are successfully decoded at the receiver. To avoid always having to explicitly define previously complete segments, a SACK carries a *cumulative acknowledgment*, which identifies a segment and the last decoded (in order) source block from that segment. The cumulative acknowledgment confirms the reception of all previous completed segments with sequence numbers below the indicated one, as well as the reception of all blocks (from the indicated segment) with indexes below or equal to the indicated one. This optimization is possible because segment sequence numbers are incremented monotonically and cannot be reused during a session. In the queue of the example Figure 5.7, the cumulative acknowledgment appears at the end of the first source blocks from segment 3. Encoded blocks are not acknowledged individually because the sender is expected to always retransmit any missing encoded blocks that contain the source blocks directly (the erasure codes are systematic).

SACKs are transmitted back to the sender whenever a data segment is fully decoded. Since the REB attempts to send data over maximum sized segments (whose encoded blocks reach sizes near to the MTU), a SACK is only transmitted by a receiver much less often than the rate of the arriving data packets. This has the advantage that receivers spend less network resources when replying to senders, which works well with the use of the UDP protocol at the underlying network (unlike TCP, which may transmit an acknowledgment for every received packet). The disadvantage of using this strategy is that the loss of a SACK could severely delay the senders from transmitting extra segments. To compensate for this, a receiver node transmits copies of the SACK through multiple routes (the same routes used to receive the blocks from the given segment, but in the opposite direction) in order to maximize the probability of the arrival of the SACK at the sender.

The REB implements a few simple rules regarding the management of the receive queue and the return of acknowledgements:

1. The receive queue places the blocks/segments in their expected position with respect to their identifiers.

2. The receive queue does not store blocks/segments beyond its maximum size. Therefore, out of bound packets are dropped.

[3]However, as the flow control mechanism is used exactly to prevent these packets from being transmitted, this can only occur if the sender is malicious.

3. The receive queue never garbage collects blocks from the segment that is currently being received, and that have been acknowledged to the source.

4. The receive queue can garbage collect blocks and segments that are beyond the segment currently being received, even if they had been acknowledged to the source.

5. Selective acknowledgments may "rollback" information about previously acknowledged segments due to the garbage collection at the receive queue.

6. Selective acknowledgments may only confirm the reception of part of blocks/segments in the queue as there is limited space in a SACK packet. The blocks/segments that should be acknowledged are the ones that appear first in the receive queue.

In the example of Figure 5.7, the segment that is currently being received is segment 3. Therefore, its blocks will never be garbage collected. The saved blocks for segment 4 or even segment 5 may be removed if the node needs to reclaim their space. Section 5.5.7 addresses the flow control mechanism which discusses in more detail the issue of garbage collection at receive queues.

SACK packets may also be transmitted back to the sender under different circumstances. For instance, retransmissions, the flow control mechanism or the route probing mechanism (see Section 5.5.8) may trigger the transmission of SACK packets by the receiver, even when no new segment had been decoded.

Retransmission After a data segment is transmitted for the first time, it is placed by the sender in a retransmission queue, where it stays until a received SACK informs the sender of its successful delivery by the receiver. The retransmission queue also stores the acknowledged status of each segment, and this knowledge is used by the sender to manage segment retransmissions. The REB manages one retransmission queue per destination at the sender.

After a segment is placed in the retransmission queue, the sender node starts a timer with a predetermined duration (in the flow control section, we will look into the case where multiple segments are placed in the queue). Ideally, the timer duration is defined in such a way that allows the receiver enough time to decode the segment and return the acknowledgment. Figure 5.8 shows an example of a transmission in this normal scenario. In this case, N encoded blocks from a segment are transmitted, and the receiver is able to successfully recover the original segment by receiving only K blocks (potentially, $K = k$ if none of the first k blocks are dropped – the erasure codes are systematic).

The arrival of a SACK packet at the sender typically removes the transmitted segment from the retransmission queue and stops the timer. However, this only happens if the segment in question is fully acknowledged by the SACK, specifically if the segment's sequence number is covered by the SACK's cumulative acknowledgment. Selectively acknowledged segments do not stop the retransmission timer, and this includes segments that only have some of their source blocks acknowledged, or segments that are fully decoded but are unordered at the remote receive queue. Selectively acknowledged information is nevertheless stored by the sender node in the retransmission queue.

If the timer expires before a SACK fully acknowledges the transmitted segment, then the network or the receiver had a problem. For example, some of the data packets were dropped and the segment could not be reconstructed, or there were delays in the network and/or receiver that made the acknowledgment arrive later than expected. Since the sender does not know the cause of the problem, it sends a special packet with no data (referred to as a *ping packet*) to the receiver to ask about which source blocks have been received so far. When the receiver node receives a ping, it immediately sends a SACK back to the

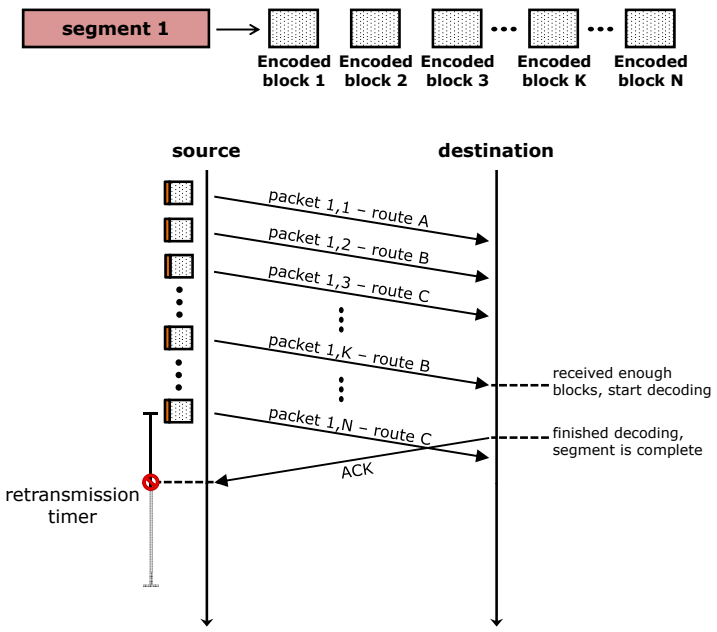

Figure 5.8: The common scenario for transmissions.

sender. The ping packets are transmitted periodically, until either a SACK arrives or the sender gives up (and returns an error). With the reception of the SACK, the sender node starts to retransmit the missing encoded blocks (corresponding to source blocks), along with some potential extra blocks to tolerate new losses. Figure 5.9 shows an example scenario of a retransmission caused by lost packets. In this case, the receiver did not receive enough (K) encoded blocks for successful decoding of the segment.

Managing the retransmission timer In TCP, the value of the retransmission timer (the timeout) is calculated based on the Round Trip Time (RTT) of the communication channel[85]. TCP keeps an estimation RTT_{est} of the RTT by averaging over time independent RTT samples, and keeps a *variation* RTT_{var} of that estimation, also averaged over time. The timeout is then calculated as (G is the granularity of the timer):

$$Timeout_{TCP} = RTT_{est} + max(G,\ \ 4 \times RTT_{var})$$

In the case of the REB, one needs to account for multiple overlay routes, which have independent RTT values. Furthermore, unlike TCP where acknowledgments are transmitted by the receiver as soon as individual packets arrive, in the REB the receiver must decode a segment before transmitting the SACK, which may take a non-negligible amount of time, with varying values according to the size of the segment. We decided to take a conservative approach for the calculation of the timeout. First, we use the RTT values of the slowest route where packets were transmitted. Second, we estimate the decoding time of the transmitted segment by measuring the time it took to encode the transmitted blocks and multiplying that time by a constant value[4].

[4]Here we are assuming that the decoding process takes a time roughly equal to the total encoding time, scaled by a constant factor. This factor is obtained by taking into account the type of erasure codes we use, particularly the time complexities of both encoding and decoding procedures. As we get more experience in using the REB, we may devise a more accurate way to make this estimation.

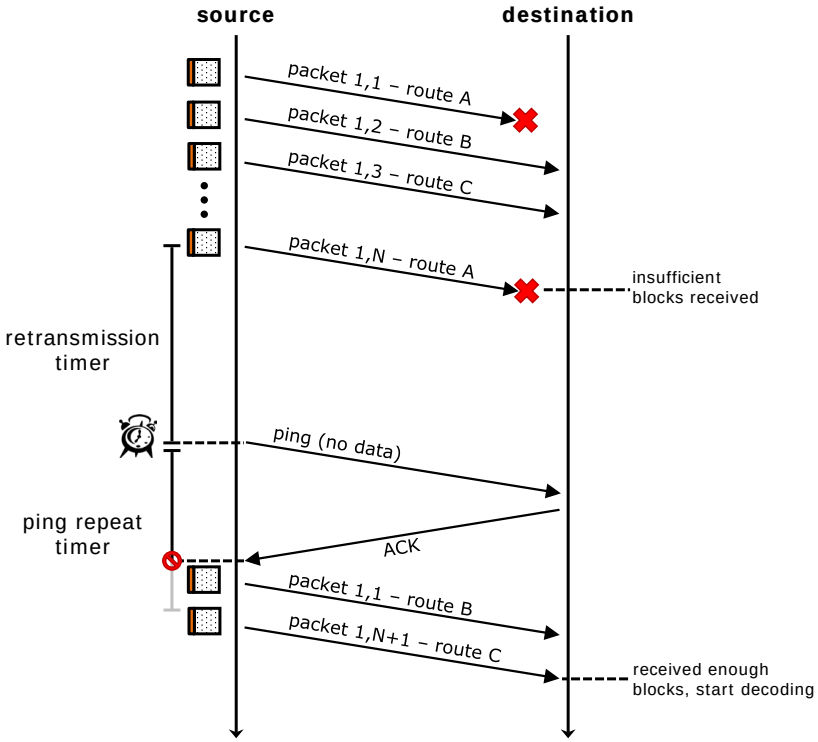

Figure 5.9: A retransmission caused by a high packet loss.

In a similar way to TCP, the REB keeps an RTT estimation and RTT variation per overlay route (Section 5.5.8 explains how these are obtained). Let RTT'_{est} and RTT'_{var} be the values for the RTT estimation and RTT variation, respectively, of the slowest route used in the transmission of the segment (i.e. the route with highest $RTT_{est} + RTT_{var}$ value). Let also δ be the estimated processing time for the decoding of the segment at the receiver. The timeout for the retransmission timer is calculated as (G is the granularity of the timer):

$$Timeout_{REB} = RTT'_{est} + max(G,\ \ 4 \times RTT'_{var}) + \delta$$

Security considerations To circumvent DoS attacks at the receivers, a receiver node only responds to pings that span over a time period at least as great as the ping sending period at the senders (which is constant).

Additionally, there is also an issue concerning the loss, duplication and/or reordering of SACK packets, either of accidental or malicious nature. For example, an attacker may try to replay previous SACKs to a sender node, trying to force unnecessary retransmissions. Even in a non-malicious scenario, the reordering of two SACK packets could make the source think that some of the previously received packets were garbage collected, therefore, they would eventually need to be retransmitted. To solve this sort of problems, when a SACK arrives the sender compares the acknowledgment information in the new SACK with the last accepted SACK, with regard to (a) the value of the cumulative acknowledgement, and (b) the list of indexes of correctly received source blocks in the segment that is currently being received. If any of the two indicates that less packets have arrived, the new SACK is discarded. Otherwise, the sender updates the current knowledge of what has been received according to the new SACK.

5.5.7 Flow control

When a sender REB node transmits data to a receiver, it may have several segments ready for transmission. In this case, it would be interesting if the node could disseminate packets from multiple segments before receiving an acknowledgment for the initial one. This has the benefit of allowing sender nodes to do useful work while the receivers are busy decoding segments, or while the SACK is being forwarded through the network. However, the increased transmission capability needs to be bounded, otherwise too many packets may reach the receivers, which can cause many packet drops to avoid memory exhaustion. Additionally, the transmission should be done in an efficient way that minimizes the overhead per segment in order to avoid, for instance, the creation of many small segments, which do not produce packets with an ideal size (close to the MTU of the network). To address these problems, the REB implements a flow control mechanism to limit and adjust the transmission rates between nodes.

Flow control at the receiver The erasure codes used by the REB are rateless, which means that senders could, in theory, generate new encoded blocks in a limitless way in order to tolerate very high loss rates in the network without recurring to retransmissions (see Section 5.5.5). However, the REB monitors the used overlay routes in order to infer loss rate values, which provide an estimated upper bound N for the number of transmitted encoded blocks (see Section 5.5.8). Nevertheless, it may happen that at some point all the available overlay routes are performing very badly, or a malicious sender node is transmitting an endless amount of encoded blocks. For these reasons, the REB imposes an absolute limit N_{max} on the number of unique encoded blocks that a sender may generate. Since encoded blocks are uniquely identified by sequence numbers (see Section 5.5.4), a receiver node will only accept the arrival of encoded blocks with numbers between 1 and N_{max}, discarding any duplicates or blocks with numbers outside this range.

The number N_{max} also imposes a limit on the *minimum* capacity (in number of bytes) of a receive queue. A receiver must be prepared to receive at least one maximally sized segment, which means that up to N_{max} encoded blocks of maximum size must fit inside the receive queue. In practice, though, a single segment should never fill the queue prematurely because the erasure codes are systematic, meaning that among the hypothetical N_{max} encoded blocks there would have to be the initial k blocks, which correspond to the original source blocks, thus effectively marking the segment as complete and forcing the receiver to discard the remaining encoded blocks.

We have seen that the rate of a transmission is dictated by the capacity of the receive queue, but more importantly, and in situations when the REB is used correctly, the rate of a transmission mostly depends on the rate of data consumption by the application. The receive queue holds (in order) the latest completely received segments until they are consumed. To tolerate out-of-order reception, encoded blocks from subsequent segments are also buffered inside the queue. If those segments are decoded before the previous ones, they stay in the queue until the earlier segments are read (in order) by the application. If the application is consuming segments at a low rate, the receive queue may fill up with complete segments and will not accept additional ones until space is freed inside the queue again.

Since the data consumption rate by an application affects the available space inside the receive queue, sender nodes should be informed of this rate in order to adjust the amount of transmitted bytes accordingly. Selective acknowledgments (or SACKs) provide information to the sender about which segments are already received (see Section 5.5.6), but do not inform about the available space for new segments inside the queue. For this reason, a receiver node appends to SACK packets a special value that provides this information. This value is named the *receive window* and is defined as the size in bytes of the por-

tion of the receive queue that does *not* contain segments ready to be delivered to the application. This portion of the queue may contain incomplete (or complete but unordered) segments that were already transmitted by the sender. Since information about those segments is included in the SACK, the sender simply subtracts their total size from the receive window in order to obtain the free space in the receive queue for any missing blocks or new segments.

Receive windows suffer from a known problem called *silly window syndrome*, that occurs when the size of the window is positive but very small (smaller than the maximum size of a packet). This makes senders transmit very small amounts of data at a time, increasing the overhead per packet. To avoid this problem, receiver nodes announce the real value of receive windows only when they are at least as large as the MTU of the network, otherwise they announce a value of 0, which effectively stops the sender from transmitting any more data until the window increases again past the minimum value. Additionally, since the initial value of a receive window would only be known to the sender after it received the first SACK packet (which would require the sender to send some data first), the receiver announces the window value inside one of the handshake packets during session establishment, which occurs before any data is exchanged (see Section 5.5.2).

Flow control at the sender A sender node keeps an informed view of the receive queue from each remote node, based on information that is returned inside SACK packets by the receiver (the view may not depict the correct remote state at all times, but it eventually converges to the right one if SACK packets continue to arrive). This information consists in the acknowledged status of each transmitted segment (which is stored inside the appropriate retransmission queue, see Section 5.5.6), as well as the latest known value of the receive window, which limits the amount of data that can be (re)transmitted.

The receive window announced by the receiver decreases as segments inside the receive queue are decoded (and can be readily made available to the application), and increases as the application consumes those segments from the queue. If the queue fills up faster than the application can consume the data, then eventually the receiver will announce a closed window (with value 0). In this situation, the sender stops the transmission of new segments and initiates a periodic transmission of ping packets (packets with a data of size 0, as described in Section 5.5.6) until a new SACK packet arrives with an increased window value.

The size of the remote receive window is also used to control the extraction of a new data segment from the respective segment queue (see Section 5.4). The sender first calculates the usable window Uw by subtracting from the remote receive window the total length of every outstanding segment that is waiting to be acknowledged (inside the retransmission queue). Based on the value of Uw, on the number of outstanding segments, and on the current amount of bytes A_{seg} inside the segment queue, the sender proceeds with the following steps:

1. Calculate $A' = min(Uw, A_{seg})$;
2. If A' is larger than or equal to the *maximum segment length*, then extract a segment with this length;
3. If A' is larger than or equal to the MTU (minus the headers) but less than the maximum segment length, then extract a segment with A' bytes;
4. If A' is small (less than one MTU minus the headers), then only extract a segment with A' bytes if there are no outstanding segments in the retransmission queue.

After a segment is extracted, it is immediately transmitted and placed at the end of the retransmission queue. To prevent the creation of many small segments, while avoiding the indefinite delay of the

transmission of application data, the procedure above only allows a small segment to be transmitted concurrently to a certain receiver.

The retransmission of a segment is conditioned as well by the latest known value of the remote receive window, along with the size of previously transmitted segments inside the retransmission queue. For every retransmission of a segment, the general procedure steps are the following (all sizes are in bytes):

- Let $window_{seg}$ be the current receive window minus the size of the total source data from every segment inside the retransmission queue that come before the current segment.
  Let $Remaining_{seg}$ be the set of bytes from the current segment that are still unacknowledged.
  Let $trans(x)$ be the function that returns the actual data to be transmitted from the input bytes x (discussed afterwards).
  Let $minspace_{seg}$ be the expected minimum space required by the receive queue to hold the current segment as it is being received (also discussed afterwards).

- If $window_{seg} < minspace_{seg}$ then:

  1. Let $Remaining'_{seg}$ be the subset of $Remaining_{seg}$ that only has those unacknowledged bytes contained within the first $window_{seg}$ bytes of the segment;
  2. Send $trans(Remaining'_{seg})$ through the overlay routes and keep a record of this transmission;
  3. If it had not been started already, start a periodic transmission of ping packets, to force the receiver node to respond with an acknowledgment.

- Otherwise, if there is space in the receive window:

  1. Send $trans(Remaining_{seg})$ through the overlay routes and keep a record of this transmission;
  2. Start the retransmission timer for the current segment (as described in Section 5.5.6).

In the procedure above, the function $trans(x)$ receives a segment (or part of it) and performs an operation over it, producing afterwards individual packets that contain the output from the operation, the headers for the data and overlay routes, and one or two MACs, depending on the type of route (direct or with an intermediary node). The operation over the input segment behaves in a different way according to the size of the segment and the conditions of the utilized overlay routes. Additionally, the value $minspace_{seg}$ also changes depending on the size of the segment:

- If the segment is large enough to be encodable by the erasure codes, then the operation corresponds to the generation of a number N of encoded blocks, where N is calculated from the combined loss rates of the routes and a number K of blocks sufficient for a high probability of decoding at the receiver (see Section 5.5.8). In this scenario, $minspace_{seg}$ corresponds to the length of K blocks;

- If the segment is too small to be encodable, that is, if its size allows it to be placed entirely inside a packet, then the this operation simply returns the input segment. In this case, $minspace_{seg}$ is simply the size of the segment.

Furthermore, in the procedure above, the calculation of the available receive window, for a particular segment, takes into account the size of *source data* from previous segments, instead of the size of transmitted data. This decision is made by the sender because it reflects the end result at the receive queue, where segments are decoded and therefore have their original sizes. It may happen, however, that decoding is not always immediately successful, or that the value $minspace_{seg}$ is insufficient to contain the incoming segment (i.e. K blocks are not enough for decoding), and thus received segments occupy more space than expected, leading sometimes to the discarding of data.

Garbage collection of segments The portion of the receive queue that starts immediately after the last in order complete segment (or the whole queue if it is empty) can be used to store further segments. The size of this portion is given by the receive window, and when managing it we give priority to the next transmitted segment because it will be read by the application right after the existing ones. Therefore, the blocks of this segment, which we call the *segment currently being received* (see Section 5.5.6), are never reclaimed by the garbage collection procedure in case of lack of memory in the node. The blocks from the following segments can, however, be deleted, and in general we give priority to the segments with lower sequence numbers, discarding if possible segments with higher numbers in the queue.

Garbage collection occurs at the receiver because the receive window includes the total size of encoded blocks per segment inside the receive queue, while the receive window perceived by the sender only includes the original size of the transmitted segments. The sender assumes optimistically that the receive window is larger than it actually is because eventually, when enough encoded blocks from a segment arrive at the receiver, the complete segment is reconstructed and any extra encoded data is removed. However, it may happen that insufficient encoded blocks reach the receiver for successful decoding (less than K blocks), but still the number of blocks at the receive queue exceeds the number of k source blocks. Alternatively, in rare occasions, K blocks arrive but are not sufficient for decoding. In any of these situations, an incomplete segment takes up more space inside the receive queue than what the sender had expected. If the additional space is not available, the receiver may discard received blocks from the next segments (and possibly even decoded segments). This is reflected on the way acknowledgments are understood by the source, where a newer SACK may "rollback" the delivery status of subsequent blocks/segments. Figure 5.10 shows an example of this scenario. Note that thanks to the efficiency of the erasure codes, it is expected that such scenarios occur rarely in practice.

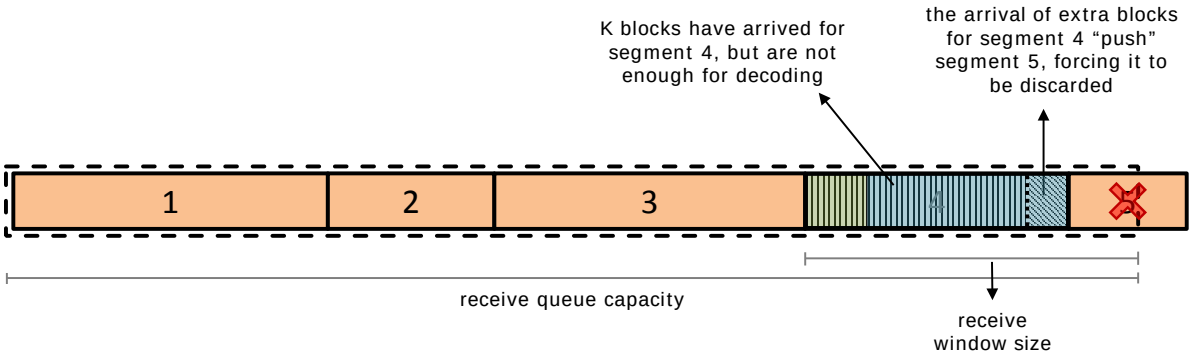

Figure 5.10: Example scenario where a segment has to be discarded from the receive queue to get space for storing blocks of the segment that is currently being received.

Security considerations A sender node only deems valid certain receive window values, particularly those that do not exceed the *receive queue capacity* of the remote node, which is fixed per session and known by the sender[5]. This is done in order to avoid DoS attacks by malicious receivers that could exhaust the memory at the senders by announcing very high receive windows.

On the subject of receive window updates at the sender, it is possible for a duplicate acknowledgment (spurious or forced by an attacker) to give an incorrect view of the window, forcing unnecessary retransmissions. This can happen when a receiver node sends acknowledgments which roll back the information about the delivery status of subsequent segments (see above). This situation arises because even though it is possible for the sender to detect an ordering on the acknowledgments by analyzing the delivery status of the first segment (which never rolls back), different acknowledgments that only differ on the subsequent segments cannot be consistently ordered. This rollback behaviour is expected to occur only on rare occasions, though, so this particular scenario does not significantly affect the communication. Besides, note that it only has a small impact on the communication progress because the first segment continues to be delivered correctly.

Similarly to retransmissions, a receiver node only responds to pings that span over a time period at least as great as the ping sending period at the senders (which is constant). This is done to circumvent DoS attacks at the receivers.

5.5.8 Route probing and selection

In order to fully take advantage of the erasure codes and to avoid data retransmissions, it is desirable to estimate upper-bounds on the quality of service of the used overlay routes. Therefore, it is required to periodically monitor the state of these routes, in order to have a continuously informed and updated knowledge about the best available paths.

Concerning metrics of quality of service, a common one is the Round Trip Time (RTT). For a sender node, it is both important to know how much time will take for a given data segment to reach its destination, and also how long it will take before the respective acknowledgment arrives back, so that the retransmission timer can be adjusted accordingly and expire only rarely (see Section 5.5.6). Furthermore, the application benefits if their data is delivered as soon as possible to the destination, so the REB selects the overlay paths with lowest RTT to disseminate the data.

Another metric is the loss rate of a route, which influences the effectiveness of the communication. A lost packet means the data carried inside must be transmitted in another packet, either using (a) a replicated copy, (b) delayed retransmission or, in our case, erasure codes that add redundancy, requiring less transmissions than (a) and less additional time than (b). By measuring the loss rate of the overlay routes, the REB can adjust the amount of redundancy per transmission, resulting in a more efficient communication. Furthermore, with this information, the amount of packet loss can be minimized by selecting the routes with lower loss rates.

Probing strategy A source node maintains a quality metric per overlay route that is affected by the respective RTT and loss rate values. The routes with the best metrics are those employed in the main communication (we call this set of routes the *active set*). In order to estimate this metric, the source utilizes a probing mechanism that is activated periodically, causing the destination node to return back

[5]Currently, the REB keeps this capacity fixed for every session, since it is not configurable by the application. In the future, if we allow it to be configurable, the receiver would only need to transmit its value in one of the handshake messages, along with the value of the receive window (see Section 5.5.2).

information about the delivery delays and packet losses. Given the current size of SIEM deployments, the REB overlay may have a few hundred nodes. To keep the overhead of probing small, we only probe a subset of all possible routes and adjust the probing frequency to the usefulness of each route — routes that are used regularly are probed more often. Consequently, a route that is used recurrently to transmit data will have more probing traffic being conveyed by the destination. Routes that are never utilized are only checked when the source decides to send a probe through them. This allows for a fast recovery of the routes in the active set (should they suddenly become attacked), while keeping the knowledge about idle routes updated over time.

A source node starts the probing mechanism immediately after a session is established with a destination (see Section 5.5.2). Two types of probing algorithms are employed to manage the traffic efficiently: a passive and an active algorithm.

Passive probing A source node sends probe requests through idle routes to trigger immediate replies, through the same routes, with probing data by the destination node. The frequency of this transmission depends on the quality of the route: good routes are probed more frequently than worse ones.

Each probe request includes a unique sequence number in it, and the respective probe reply contains that same number. This way, a source node can identify which probe is being responded by the destination. The sequence number is represented as a 64-bit unsigned integer, that starts at the value 0 and is incremented monotonically per transmitted probe. The transmission of probe requests stops when the sequence number reaches its maximum value, requiring a restart of the communication session, but considering the value is $2^{64} - 1$ and the frequency of the probing algorithm is in the order of seconds, in practice we do not expect that to happen under normal circumstances.

Active probing When transmitting data though the active set of routes, each time a segment is fully received, a destination node sends probing data back to the source, through the same routes used by the latter to transmit the segment.

Sending probing data only when segments are completely received, as opposed to transmitting this data when packets arrive, could at first seem to affect the precision of the quality estimation at the source. Adding such transmission every time a packet is received, though, would increase the network traffic in the return routes and defeat the purpose of using erasure codes. It was thus necessary to find a mechanism that kept the extra traffic to a minimum, but at the same time offered a fine granularity on the probing of individual packets (information about latencies and losses). To achieve this granularity during a normal transmission, a destination node records information on the arrival of received packets and accumulates that information on probe reports that are transmitted to the source inside SACK packets (see Section 5.5.6). At the same time, a source node records information on the departure of each packet it transmits, including the identification of the route that was used for the transmission. When a SACK packet arrives back at the source, the node compares the information reported by the destination with the local one, and updates the quality metric of the routes per packet in batch.

It should be noted that eventually, at a destination node's receive queue, the combined length of the acknowledgment and the current probing report for some particular route may reach the size of the MTU (excluding headers). When this happens, the destination node is required to transmit the acknowledgment and the probing report back to the source, even if no segment was just decoded. The REB tries to mitigate this scenario by compressing, to some extent, the probing information inside the SACK packet.

Measuring the Round Trip Time (RTT) The way the REB calculates and keeps an expected RTT value per route is built upon the well tested algorithms employed by TCP [85]. The expected RTT value corresponds to the sum of an RTT estimation and an RTT variation. Both metrics are averaged following an exponential weighted moving average, which keeps a smoothed value (compared to the actual RTT samples) over time. For each route, both metrics are computed as follows:

- If no sample RTT values have been taken, then:
  - $RTTvar = 0$
  - $RTTest =$ a predefined large value
- When the first sample RTT value has been taken, then:
  - $RTTvar = RTTsamp/2$
  - $RTTest = RTTsamp$
- When an additional sample RTT value has been taken, then:
  - $RTTvar = (1 - \beta) \times RTTvar + \beta \times |RTTest - RTTsamp|$
  - $RTTest = (1 - \alpha) \times RTTest + \alpha \times RTTsamp$
  - Note that here, $RTTvar$ is updated using the value of an $RTTest$ from the previous update
  - Following the advice in [85]: $\alpha = {}^1/_8$ and $\beta = {}^1/_4$

Taking an RTT sample per packet depends on the type of probing algorithm that is being used on a particular route. If we are using passive probing, then the source node records the departure timestamp $departure_{req}$ of a probe request with a given sequence number. When the respective probe reply arrives, the source obtains the timestamp of its arrival $arrival_{reply}$ and calculates the RTT sample as:

$$RTTsamp = arrival_{reply} - departure_{req}$$

If we are using active probing, then the source node records the departure timestamp $departure_{pack}$ of each transmitted data packet through a given route, while the destination node records the arrival timestamp $arrival_{pack}$ of the same packet and includes it in the probing report that is transmitted inside the SACK. Upon the arrival of a SACK, the source calculates the RTT of each packet using both timestamps. However, since REB nodes might not have their clocks synchronized with very high precision, the source cannot simply subtract the local timestamp from the remote one. To circumvent this problem, the destination reports an extra timestamp $departure_{SACK}$ recorded just before the SACK packet is transmitted, while the source records the timestamp of the SACK arrival $arrival_{SACK}$. Then, for each reported packet, the RTT sample is calculated as:

$$RTTsamp = (arrival_{SACK} - departure_{pack}) - (departure_{SACK} - arrival_{pack})$$

It is important to refer that each SACK packet must be transmitted through all the routes from where the reported packets were transmitted by the source. In turn, for every SACK copy it receives, the source node must only measure the sample RTT values from the packets that were transmitted through the same route as the SACK. This is required because otherwise it would be possible for the latency of a route to affect the estimation of the RTT of another. Figure 5.11 shows a graphical example of RTT sampling during transmission of data.

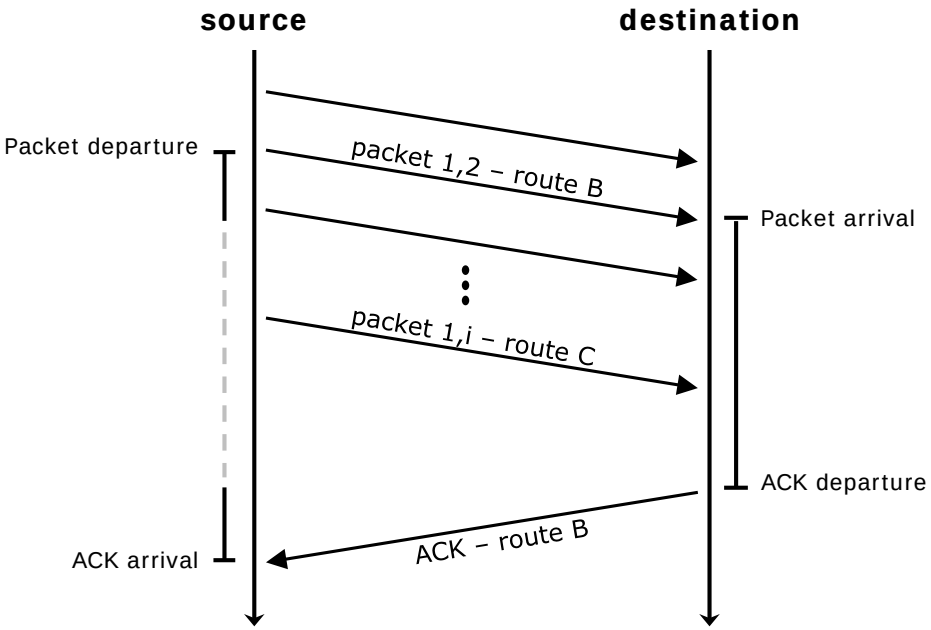

Figure 5.11: Example scenario where the RTT is sampled for packet with ID (1,2).

If a route suddenly starts dropping many packets, then the probing algorithm will not be able to measure RTT samples from transmitted packets. The same can happen if a confirmation (a probe reply or a SACK packet) is dropped. To handle this issue, the REB employs different strategies depending on the type of probing algorithm that is being applied to a particular route.

In passive probing, the source node starts a timer for every probe request transmitted through a given route such that, if it expires before any probe reply arrives, then the expected RTT value of the route increases exponentially with a predefined factor[6]. This results in the quality metric of the route becoming worse, which might remove the route from the active set, giving place to another with a better metric.

In active probing, the source node starts a similar timer for each route in the active set, every time it initiates a transmission of data. If a timer expires before a SACK packet arrives from the respective route, the source temporarily initiates the passive probing algorithm for that route. Here, we cannot simply directly affect the metric of the route, because a missing SACK might have resulted from other routes dropping data packets.

Measuring the loss rate The loss rate of a route is a rational number between 0 and 1, an estimation of the fraction of transmitted packets that are lost. Its usefulness is in enforcing a lower-bound on the number of encoded blocks, from a data segment, that should be transmitted over the routes in the active set.

Unlike the expected RTT of a route, the loss rate is adjusted differently over time depending on the used probing algorithm and whether a new rate sample increases or decreases the current value. If we are using active probing and the new sampled rate is larger than the current rate, then the adjusted rate becomes the sampled one, which means that a loss rate "jumps" to the new value if it aggravates. This has the property of quickly adapting to increases in the loss rate of a route inside the active set, which

[6]Here, the timer has a timeout value that is calculated in the same way of a data retransmission timer (see Section 5.5.6).

may result in some of those routes, that suddenly experience higher loss rates, to exit the set entirely, giving way to other routes with lower rates. On the other hand, if we are using passive probing or if the new sampled rate is smaller than the current one, then the rate is adjusted smoothly according to an exponential weighted moving average that converges the new rate to the sampled one more slowly. This means that routes with high loss rates take more time to "heal up" should they suddenly behave better, but it also means, more generally, that passive probing has less impact on the loss rate estimation (since it is not as useful in this case as active probing).

Sampling loss rate values also differs in method according to the probing algorithm. In passive probing, if a probe request, transmitted by a source node through a certain route, receives a reply from the destination (one that matches its sequence number), then the loss rate of the route is adjusted using a sample of value 0. Should the probe request or the reply be dropped by the network instead, then it is possible to detect the loss before a timeout occurs (see above in RTT) if a subsequent reply arrives in the meantime. In this case, the source looks at the sequence number inside the latest reply and compares it with the ones from requests that still await a response. If any of them is smaller than the received number, then the respective request is deleted (forgotten) and the loss rate of the route is adjusted using a sample of value 1. In both cases, the loss rate converges slowly to the new value. Note that the reordering of probe requests/replies may be perceived by the source as a loss.

Sampling a loss rate value in active probing requires that the source node inspect the IDs from the packets reported in the probing information announced by the destination inside a SACK packet. If there are "holes" within the listed IDs, the source assumes the respective packets were lost in transit. More specifically, the source assumes that every unacknowledged packet which has an ID inferior to the highest one indicated in the SACK is lost. This means that the loss of a SACK packet can severely affect the perceived loss rate of a route. However, since SACK packets are replicated through multiple routes by the destination, the chance of all being lost is reduced. The source obtains a loss rate sample $rateSamp$, from each received SACK, by checking the number of transmitted packets $total_{sent}$ and counting the number of reported packet losses $total_{lost}$:

$$rateSamp = \begin{cases} 0 & \text{if } total_{sent} = 0 \\ \frac{total_{lost}}{total_{sent}} & \text{if } total_{sent} > 0 \end{cases}$$

Route selection for data transmission A source node manages the overlay routes by ordering them using a quality metric. This metric takes into account both the expected RTT and loss rate of a route. To calculate this metric, the source uses the expected RTT as a base value and increases it linearly a given amount if the loss rate is positive. This has the property of letting both RTT and loss rate affect equally the quality metric, while still providing a useful way to order multiple routes (by RTT) when they are performing well (i.e. without incurring losses). The active set of size num_{act}, containing the best routes used for transmitting data, corresponds to the num_{act} routes with lowest metrics, alternating also between the best direct routes and the best intermediary routes (those using an intermediary node) in order to provide some degree of diversity.

Right after establishing a session with a destination node, a source chooses randomly, from all the available routes, up to r_{dir} direct routes and up to r_{int} intermediary routes, to create a set of usable and monitorable routes. Usually, $r_{int} > r_{dir}$ if sufficient intermediate nodes are configured. Also, num_{act} should be much smaller than the number of usable routes ($r_{dir} + r_{int}$), but this depends on the size of the configured network. In situations where packets are transmitted without an active session (e.g. during the handshake protocol) the used routes are chosen randomly.

Since the size of the network may result in a total of available routes much larger than the size of the usable set, the source node periodically replaces the worst routes, i.e. those with highest metrics, with other routes chosen at random (that are not already selected). This has the advantage that bad routes cease to be probed in favour of others that may perform better. Additionally, the size num_{act} of the active set may not always be fixed, although it has an upper bound max_{act}. When choosing the num_{act} routes to be used for data transmission, the source node first chooses the route r_1 with the lowest metric value. Then, up to $max_{act} - 1$ additional routes are chosen (alternating between direct and indirect) with the condition that each has a metric value no greater than $metric_1 + \lambda$, where λ is a preconfigured constant. This "fix" is applied to keep the communication stable when the number of good routes is low, which may happen if the size of the configured network is small, or if most of the routes are compromised.

Using the routes for transmission The routes in the active set are used to transmit the data in a different way according to the type of segment. Small segments that fit inside packets are replicated through every route in the set. Larger segments are handed to the erasure codes for generating encoded blocks, and each of these blocks is transmitted inside an individual packet[7] through one of the routes in the set, alternating between routes in a round robin fashion.

The number N of transmitted encoded blocks is influenced by the loss rates of the routes. Given a number K of encoded blocks (a number of blocks required for a high probability of successful decoding at the destination), the average loss rate $avgRate$ of all the routes in the active set, and the maximum number of encoded blocks N_{max}:

$$N = \begin{cases} N_{max} & \text{if } avgRate = 1 \\ min(\lceil \frac{K}{1-avgRate} \rceil, N_{max}) & \text{if } avgRate < 1 \end{cases}$$

Note that N may still be insufficient if one or more of the routes in the active set suddenly starts dropping packets. In this case, the metrics of the affected routes would only be updated after a timeout occurred, and the destination might not have received enough blocks in the meantime for a successful recovery of the segment. To circumvent this problem, the REB may be optionally configured to increase the number N a certain amount, such that even if a number f of routes are suddenly attacked, enough packets reach the destination. This increase is still subjected to the same upper bound N_{max}, though.

Security considerations An attacker that takes control of one of the overlay routes may attempt to deceive a source node by making it think the route is a good one. For instance, the attacker may decide to forward the probing traffic, and discard only the data packets. However, if the route is actually used for data transmission, then the probing traffic is included in the main communication one (inside SACK packets), which makes the traffic separation impossible.

If an attacker takes control of all the routes in the active set, she may suddenly drop all the transmitted packets, delaying the detection of the problem by the source because the node is forced to wait for a timeout. However, given the power of the attacker, there is not much we can do in this case, and communications will be temporarily interrupted. To mitigate this problem, the REB may be configured to have a larger set of active routes.

Concerning passive probing, an attacker may attempt to perform a DoS attack at some destination node by transmitting a large number of replayed probe requests. The destination solves this issue by recording the sequence number of the last probe request it responded to, while refusing to respond to any request with a number lower than or equal to the last. A malicious source node may still try to

[7] If the blocks are small enough, a packet may carry multiple blocks (more details in Section 5.5.5).

perform the same attack, circumventing the previous restriction by generating a large number of requests with increasing sequence numbers. The destination addresses this problem by refusing also to respond to more than a constant number of requests from a given source per second (which can be safely upper bounded since the frequencies of the passive transmissions, as well as the maximum number of probed routes by a source at any time, are known by every node).

5.6 Experimental evaluation

This section presents an experimental evaluation to assess some of the performance characteristics of the REB, when compared to standard solutions available in current systems, namely the ones based on the Transport Layer Security (TLS) protocol and on the UDP over the IPSec protocol. The tests measured the latency times of the communications between a sender and a receiver node, with the quality of the communication links (i.e., the loss rate) varying on each test run.

The results show that under optimal network conditions, the REB is capable of performing in a comparable way as the other protocols, without introducing noticeable overheads. Under poor conditions, where certain network links experience packet losses (of 5% and 10%), the REB maintains an efficient communication, confirming that a low replication overhead provided by a combination of multipath and erasure codes effectively overcomes communication faults in the network.

5.6.1 Implementation and testbed

The experiments were performed in a controlled environment, in a system where a few PCs with homogeneous characteristics are interconnected to a local network. The LAN contains two switches, on top of which 2 disjoint VLANs are configured. The switches are HP E3500yl-24G-PoE+ and are setup to run at 1 Gbit/s. Each of the PCs contain 32 GB of RAM and two Intel(R) Xeon(R) CPU E5520 at 2.27GHz, with each CPU having 4 cores (allowing a total of 16 virtual cores due to hyper-threading). Each PC also contains 2 physical network interfaces, which connect to two different VLANs.

For our experiments, one of the PCs represented a sender and another a receiver. Depending on the test, we used either one or two network interfaces on each PC, in order to configure multiple direct routes for the REB. Both sender and receiver PCs ran Ubuntu 10.04 over a Linux kernel with version 2.6.32-21-server. Additionally, since the REB is implemented in Java, we executed the REB nodes with an Oracle's Hotspot JVM v1.7 and a configuration to allow a faster JIT (Just in Time) compiler bootstrap, to ensure that the benchmark code was fully compiled to native code. We also used the same JVM configuration to run tests for TLS and UDP+IPSec, where a simple Java application was employed to exchange data with sockets based either on TLS/TCP or UDP (with IPsec set in the host).

In order to vary the quality of a communication link between the sender and the receiver, we resorted to a network tool called dummynet[8]. This tool supported the creation of rules that configure a network loss rate at a certain host, for traffic with a particular source and destination addresses. The tool works by intercepting incoming packets on the receiver and, if the addresses in the packet match the ones in the rule, dummynet chooses randomly whether to discard the packet or not, according to the configured probability (over a uniform distribution).

[8] Available at: http://info.iet.unipi.it/ luigi/dummynet/

5.6.2 Benchmark description

We chose to evaluate the latency of the communication to determine if the REB is capable of delivering events in a timely way, despite a percentage of the transmitted packets being dropped before reaching their destination. The tests were also replicated under the same conditions with other network protocols (TLS or UDP+IPsec).

A benchmark measured the communication latency between two hosts. To accomplish this, we programmed a sender to transmit messages with various lengths and programmed a receiver to respond to each arriving message with another one of equal length back to the sender. This process allowed the sender to collect a Round Trip Time (RTT), as it measured the elapsed time from the moment just before transmitting a message to the instant just after receiving the response. The procedure was repeated a large number of times in order to get a significant sample. At the end, the sender calculated the average RTT and divided it by two to obtain the average latency time.

We also made sure that the sender would only start collecting RTT values after a predefined warm-up time (in which messages were transmitted without collecting times), to ensure that protocols had enough time to initialize the connection, as well as to ensure that all Java code was properly compiled by the JIT.

Three different groups of benchmark tests were performed, each with a distinct configured packet loss probability at the receiver. The first group of tests measured average latency times between a sender and a receiver under optimal network conditions, i.e., with no packet loss. For this first group, we ran tests for the REB configured with a single route between the sender and receiver; for TLS (v1.2); and for UDP over IPSec (setup with the Authentication Header protocol). In the second and third groups of tests, we forced a particular packet loss probability in one of the links and the following experiments were carried out: the REB used only one route between the hosts; the REB utilized the two routes; and the TLS test. These tests did not evaluate UDP+IPsec because UDP is unable to recover from the missing packets.

5.6.3 Performance results

For each set of benchmark tests, we obtained average latency time for messages with length of 1, 10, 100, 1371, 10.000, 65.483, 100.000 and 1.359.000 bytes. This allows to observe the behavior of the protocol for message lengths at different levels of magnitude. Three length values are specially relevant: 1371 bytes is the maximum length of a REB segment that fits inside a network packet (and is thus not processed with erasure codes but the data is simply replicated); 65.483 bytes is the maximum length of a UDP packet, taking into account the sizes of IPv4 and IPSec AH headers; 1.359.000 bytes is the maximum length of a REB segment in the current implementation.

Figure 5.12 displays the results of the first set of benchmark tests that were run under normal network conditions (with no losses). The graph shows that latency values of the REB are not significantly different from those of TLS and UDP+IPsec. Sometimes the REB outperforms TLS, while in other cases TLS has a smaller latency (notice the graph is in a logarithmic scale in both axis). On the other hand, UDP+IPSec consistently outperforms the REB, although on long messages the REB almost reaches the same latency value [9].

Given that the REB runs as a library in user space and not in the kernel, we found these latency results very promising. Recall that UDP+IPsec makes no effort to ensure reliable communication, and

[9] Note that it was not possible to transmit larger UDP messages than 65.483 bytes and so the curve for UDP+IPSec terminates earlier.

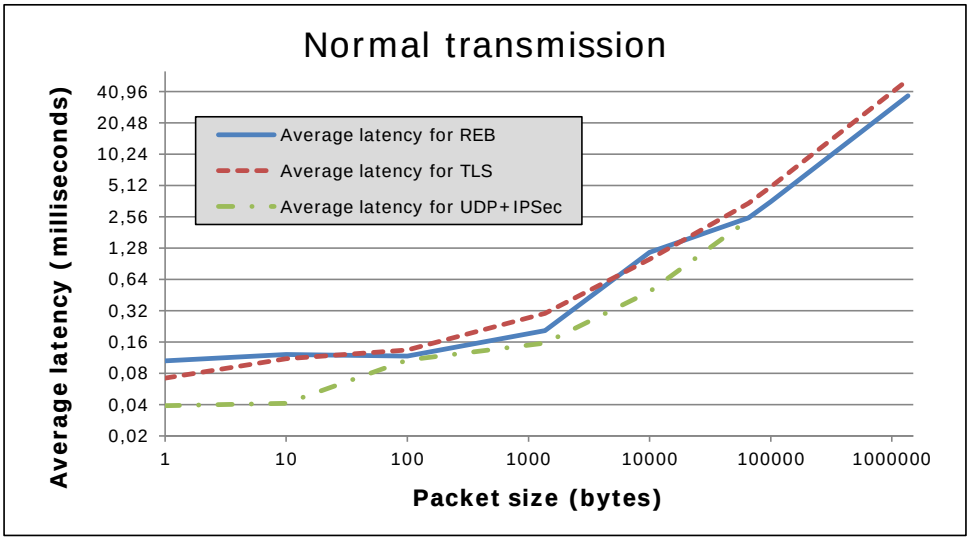

Figure 5.12: Results for a benchmark test with normal network conditions.

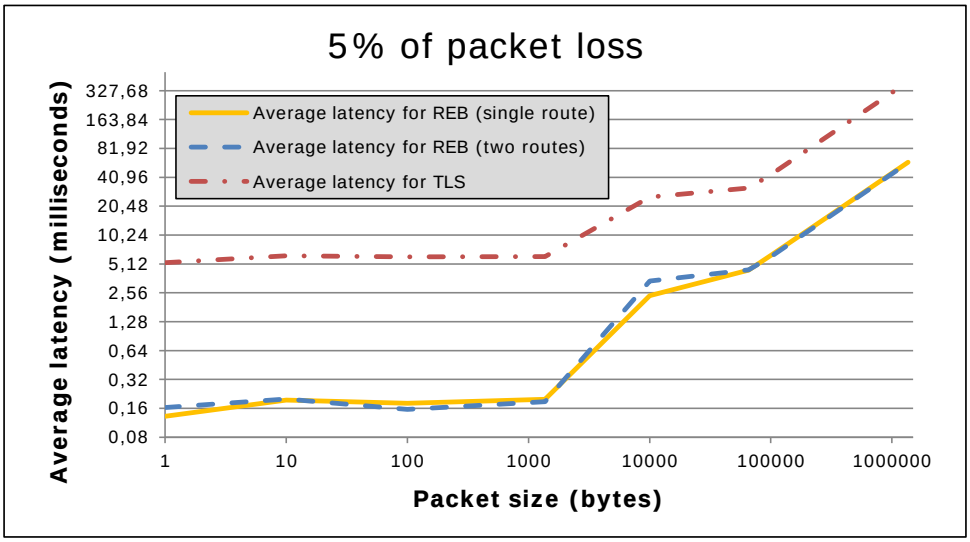

Figure 5.13: Results for a benchmark test with a link with a packet loss probability of 5%.

therefore, it is expected that some overheads can be avoided (e.g., keeping certain timers). Moreover, although some effort has already been done to optimize the current version of the REB, there is still space for improvement. For example, due to limitations of the socket Java interface, sometimes the REB is required to perform a (unnecessary) copy of the data to be transmitted. Currently, we are working on ways to eliminate this sort of problems, exactly to reduce the delays in the critical transmission path.

Figure 5.13 displays the results for the second set of benchmark tests, where one communication link was configured with a packet loss probability of 5%. The results for a similar experiment are shown in Figure 5.14, but now with a loss probability of 10%. In both graphs, the REB consistently shows a significantly better latency than TLS. TLS is built over TCP, and consequently relies on TCP algorithms to recover from missing packets. Two main problems affect the performance of TLS: first, TCP only

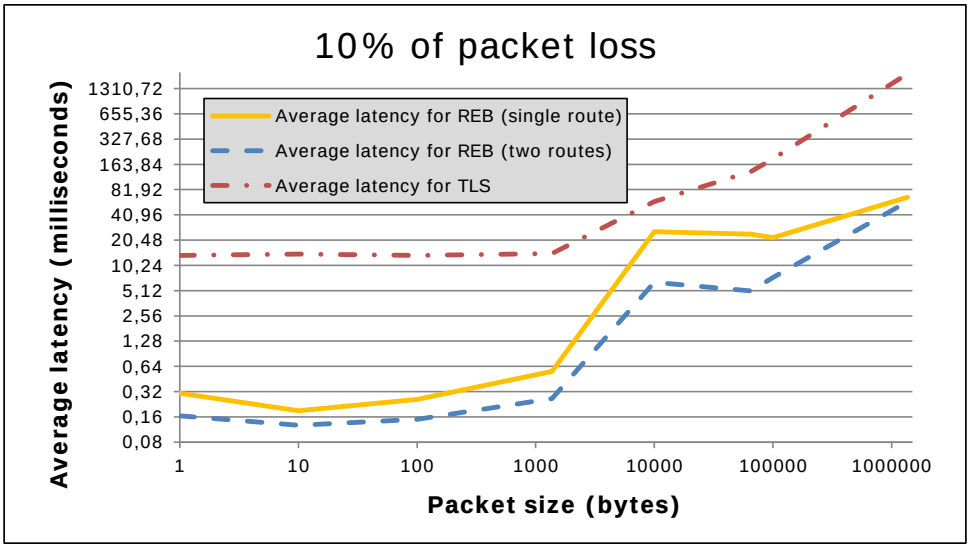

Figure 5.14: Results for a benchmark test with a link with a packet loss probability of 10%.

uses retransmissions to ensure that all data is delivered to the receiver; second, the congestion control algorithm of TCP is highly affected by packet losses, causing the transmission window to become very small. In this experiment, the REB takes advantage of two mechanisms to recover from the losses. It adds redundancy to the information being sent and, whenever possible, takes advantage of the available multiple routes to disperse the packets over different parts of the network (and therefore, reduce the effects of specific links with problems).

As the graphs demonstrate, the two mechanisms are needed to address the failures. It is interesting to notice that the usage of two routes in the first graph does not decrease the communication latency over a single route, while on second graph one can see an improvement. This happens because a loss rate of 10% becomes important enough to make the usage of an extra good route useful to avoid unnecessary retransmissions that may occur when employing only one route.

Looking at the REB curves in all graphs, we can see that sometimes there is a greater increase in the average latency between 1371 and 10.000 bytes. This happens to be a point where the REB goes from sending whole segments of data inside packets, to encoding segments into multiple blocks and transmitting those blocks inside different packets. This seems to indicate that further tuning is needed to decide when the codes should start to be employed, as we can see a more efficient utilization of the codes with larger segments.

6 Node Defense Mechanisms

This chapter addresses defense mechanisms that can be employed to enhance the dependability and security of the nodes. These mechanisms start to be presented in a generic way, as some of them might be appropriate for certain nodes while others to other nodes. Moreover, given the various criticality levels of the SIEM nodes and the cost of using some of these techniques, it is worth to consider a hierarchy of designs that are incrementally more resilient. We employ replication techniques to ensure correct node behavior in the presence of component failures of either of accidental nature or forced by an adversary. These techniques support the construction of *Intrusion-Tolerant Systems*, allowing failures to be automatically recovered as long as enough good replicas remain operational.

To put these ideas in practice, the chapter also presents the design of a core-MIS that acts as an application-level firewall to defend the core facility components. Since all traffic to and from the core facility needs to go through the core-MIS, it can perform various filtering operations to prevent external malicious messages from entering. The core-MIS employs a two level filtering scheme to increase performance and to allow for some flexibility on the selection of fault-tolerance mechanisms. The first filtering stage is designed to eliminate the most common forms of attacks, while the second stage supports application rules for a more sophisticated analysis of the traffic. The fault tolerance mechanisms are based on a detection and recovery approach for the first stage, while the second stage uses Byzantine state machine replication and voting.

6.1 Resilient mechanisms support

This section starts by providing an overview of intrusion prevention and tolerance, and then describes four mechanisms that can be used to enhance the resilience of a node: Byzantine fault tolerance, diversity, proactive recovery, and a combination of proactive-reactive recovery.

6.1.1 Intrusion prevention

The advancements in software development have provided us with an increasing number of useful applications and services with an ever improving functionality. These enhancements, however, are achieved in most cases with larger and more complex projects, which require the coordination of several teams. Third party software, such as components off-the-shelf (COTS), is frequently utilized to speed up development, even though in many cases it is poorly documented and supported. This affects the quality of the software, and everyday new flaws (including vulnerabilities) are found in what was previously believed to be secure applications, unlocking new risks and security hazards.

If not addressed prior to deployment, these flaws can result in accidental faults, such as when a boundary condition is violated and creates a process crash, or in intentional (or malicious) faults. An intentional fault can be caused by an external entity that performs a successful attack to exploit a vulnerability, and the impact is various, for instance, the corruption of data or the termination of a service. In any case, in the presence of faults the node can start to behave abnormally, i.e., stop to provide the expected service.

Over the years, security researchers have developed a set of common practices that help to *prevent* intrusions from occurring [51, 75]. These practices include a number of tasks that should be performed before connecting a node to a network. Example tasks require that: users select secure passwords, and that these are periodically updated; unnecessary or default accounts are removed; network services that are not strictly necessary are eliminated; and the file system is properly configured with the right permissions associated with files and directories. Furthermore, various kinds of testing tools, such as scanners, can be employed to find know vulnerabilities in the node and then support their removal. The software should also be updated with the most recent security patches, to ensure that existing exploits can not utilized to compromise the node.

Within the MASSIF projet, we have conducted some investigation on methodologies and tools that can contribute for the identification of new (or previously unknown) vulnerabilities in software components. A technique was proposed for inferring a behavioral profile of a network service, which models its correct execution by combining information about the implemented state machine protocol and the server's internal execution [9]. Flaws are then automatically discovered if the service's behavior deviates from the profile while processing a number of test cases. The construction of the behavior profile is done using a protocol reverse engineering mechanism that is capable of understanding the packets exchanged between the service and its clients [11].

We have also proposed a technique that is able to test new protocols and features of a service by recycling test cases from several sources, even if aimed at distinct protocols [10]. It resorts to protocol reverse engineering techniques to build parsers that are capable of extracting the relevant payloads from the test cases, and then applies them to new test cases tailored to the particular features that need to be checked. An evaluation with 10 commercial and open-source testing tools and a large set of FTP vulnerabilities showed that our approach is able to get better or equal vulnerability coverage than the original tools. Some work was also performed on the design and implementation of a tool that analyzes the source code of an application, and automatically finds potential flaws and inserts fixes to remove the discovered vulnerabilities [76]. The use of the tool was demonstrated with two open source energy metering applications in which it found and corrected 17 vulnerabilities.

6.1.2 Intrusion tolerance

Current practice has shown that software components can experience various types of faults, even after being thoroughly tested. One of the most efficient and transparent ways to deal with these faults is to tolerate them. Replicated systems have been used to solve this problem over the last decades, in the majority cases only addressing accidental faults. These solutions are thus suitable for settings relatively benign (or where intrusions can be dealt with by some other means, e.g., with insurance), but they easily fall prey of an adversary that is able to compromise a single replica.

A system is said intrusion-tolerant when a subset of the replicated components fail without harming the offered service (for an overview of the area see [105]). Redundancy is the cost attached to intrusion tolerance, for instance, by replicating the client's request in several replicas. Hence, each replica executes

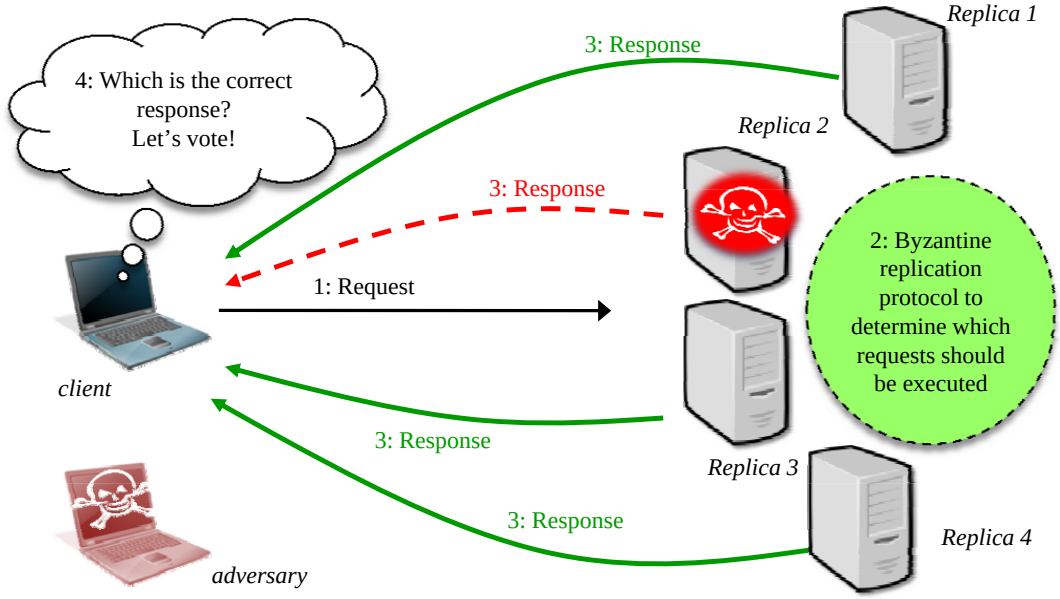

Figure 6.1: An intrusion-tolerant service accessed by a client (and potentially by an adversary) that suffers a fault in one of the replicas, but still ensures that correct responses can be obtained.

the same request and the system can assure that a number of failures can be tolerated with a (relatively simple) voting on the produced results. The number of required replicas is related to the number of faults one wants to tolerate, and its value depends on the type of service/algorithm that is implemented and on the strength of the assumptions that can arguably be made on the execution environment. Typically, the number of replicas n that is needed to tolerate f faults is $n \geq 3f + 1$, in an environment with very weak assumptions [67].

To illustrate these points, Figure 6.1 shows a intrusion-tolerant service that is accessed remotely by clients. The system offers a service to the clients under very weak failure assumptions. Typically, it is assumed that an adversary can perform various attacks on the network (e.g., replay, delay, re-order, corrupt messages), and that he or she controls an undetermined number of clients and up to f replicas. Nevertheless, even under these challenging conditions, the system needs to provide correct responses to the good clients.

The design and implementation of the intrusion tolerant system has to address two main concerns. First, the protection of the communication between the client and the service, where it is necessary that all correct replicas execute the request and then that the client is able to select a correct response. By employing re-transmissions and cryptographic methods, it is possible to secure the messages from the network attacks and ensure their delivery. The selection of the response is a bit more delicate because some of the replies might be produced by malicious replicas, and therefore contain erroneous data. Consequently, the usual procedure is to wait for $f + 1$ equal responses, and only then choose this answer (this ensures that at least one correct replica vouches for the reply).

Second, all correct replicas should start to execute from an identical initial state, and then they should evolve through the equivalent states as they process the requests (by assuring that requests are executed in a deterministic way). For this reason, a service with these characteristics is said to implement *State Machine Replication (SMR)* [93]. This is achieved by running a Byzantine fault-tolerant replication

protocol among the replicas, which associates a processing order number to each request and makes sure the correct replicas carry out the same requests.

In MASSIF, SMR brings however a difficulty due to its programming model — the assumption that there is a client that invokes an operation on a replicated service and waits for replies — since it does not match the way some SIEM components operate. For example, the core-MIS processes the message traffic to and from the core facility, and therefore, arriving messages need to be transmitted to a node other than the original sender. Nevertheless, we will describe later in the chapter the implementation of a replicated core-MIS that can act as a highly resilient application level firewall.

6.1.3 Byzantine fault tolerance

A key building block of intrusion-tolerant systems is Byzantine fault-tolerant (BFT) protocols, which guarantee the correct behavior in spite of arbitrary faults (also called Byzantine faults), provided that a minority of the components are faulty.

In our implementation of a BFT replication protocol, we are considering two forms communication support: reliable and atomic broadcast.

Reliable Multicast. A reliable multicast protocol is used to ensure that if a message is sent to a group of processes (or replicas), either all correct processes deliver this message or none will do that. Formally, a reliable broadcast protocol can be defined in terms of the following properties [19, 37, 57]:

- *Validity* : if a correct process broadcasts a message m, then some correct process eventually delivers m.
- *Agreement* : if a correct process delivers a message m, then all correct processes eventually deliver m.
- *Integrity* : for any identifier ID, every correct process p delivers at most one message m with identifier ID, and if $sender(M)$ is correct then M was previously broadcast by $sender(M)$.

We consider that the sender also delivers the messages it broadcasts, and the predicate $sender(M)$ gives the field of the message header that identifies the sender.

Total order multicast. A total order multicast (TOM) protocol is similar to a reliable broadcast protocol, but it ensures an additional property [37]:

- *Total Order* : if two correct processes deliver two messages m_1 and m_2 then both processes deliver them in the same order.

This additional property makes the implementation of a total order broadcast much harder than reliable broadcast. More precisely, the problem of total order broadcast has been shown equivalent to the well known consensus problem [57, 21, 37], and thus cannot be solved deterministically in asynchronous systems with crash failures (and therefore, cannot also be solved in systems with Byzantine failures) [46]. Over the years, several approaches have been proposed to circumvent this limitation, i.e., to slightly modify the system model in such a way that consensus becomes solvable. Examples include randomization [87, 14], failure detectors [21, 72], partial-synchrony [45, 43], and hybridization/wormholes [35, 82].

On the other hand, the up side is that a significant amount of research has been carried out on the design of new consensus protocols (for a survey see [39]), and this work can be leveraged to build better total order broadcast protocols. For example within MASSIF, we have recently proposed an optimization to a well known randomized BFT consensus protocol [103], which uses a speculative execution to reduce the number of communication rounds from three to two in the normal case. We have also proposed a new consensus protocol that is able to address very dynamic environments, tolerating tolerating attacks on the network, and providing a significant improvement in performance when compared with previous solutions [79]. In another work, a specialized wormhole was used to decrease the number of replicas that is needed to support state machine replication (from $n = 3f+1$ to $n = 2f+1$) [38]. This has interesting practical implications, as it can contribute to diminish the costs of resorting to replicated solutions.

6.1.4 Replica diversity

Works concerning BFT protocols tend to assume that replica nodes fail in an independent manner [20, 112, 15, 36, 80, 1]. To respect this condition, system components need to exhibit failure diversity. However, when security is considered, the possibility of simultaneous attacks against several components cannot be dismissed. If multiple components exhibit the same vulnerabilities, they can be compromised by a single attack, which defeats the whole purpose of building an intrusion-tolerant system in the first place. Moreover, this problem is also relevant from an accidental fault perspective. Since all replicas are executing similar tasks, if the software contains a bug that is activated when processing a request, then every replica will progress into a faulty state.

To circumvent this limitation one must substantiate the fault independence assumption by construction. Diversity allows one to build safer replicated systems based on the assumption that different components exhibit independent failure modes with high probability. Replicas should execute distinct software that offers similar functionality (e.g., think about the operating system), but since the code was developed by different teams no common flaws should occur. Moreover, every diverse replica should be fully patched, clean from known vulnerabilities, to increase the difficulty of attacks.

One can find several opportunities to use diversity when setting up a system. For example, various hardware platforms could be employed and/or the software could be configured in a different way (e.g., using randomization of the placement of the programs in memory), which often is enough to limit heavily the success of attacks that are performed with automated tools. If higher levels of security are desired, then one could use multiple operating systems (OS) and other support software. As a last step, even different implementations of the applications can also be employed, especially if the replicated component has a standardized interface.

Nowadays, nearly all software systems rely on COTS, i.e., third party software components readily available for use, like graphic packages, mathematical libraries, operating systems and database management systems. This is mostly due to the sheer complexity of such components, coupled with benefits such as the perceived lower costs from their use (some of the components may be open-source and/or freely available), faster deployment and the multitude of available options. Consequently, leveraging on COTS to implement diversity is less complex and more cost-effective than actually developing variants of software.

Within MASSIF, we have been exploring the idea of using diversity at the operating system level. Realistically, people will resort to a COTS operating system rather than build their own. Given the variety of operating systems available and the critical role played by the OS in any system, diversity at the OS level can be a reasonable way of providing good security against common vulnerabilities at little extra

cost.

In a recent study, we analyzed the likelihood of common vulnerabilities on operating systems [48, 49]. We analyzed more than 15 years of vulnerability reports from the NIST National Vulnerability Database (NVD) totaling 2120 vulnerabilities of eleven operating system distributions. The results suggest substantial security gains by using diverse operating systems for intrusion tolerance. Some of the more specific findings were: the number of common vulnerabilities on the studied operating system pairs was reduced by 56% on average if the application and locally-exploitable vulnerabilities could be avoided; more than 50% of the 55 OS pairs studied have at most one non-application, remotely exploitable common vulnerability; that there are some setups for a four-replica diverse system that have experienced very few (or no) common vulnerabilities over the years.

We have also been researching ways to build diversity at the application level. In particular, we have looked into the problem of incompatibilities that are created when using different implementations of the same application in the replicas [8]. Various kinds of incompatibilities were analyzed, and a new methodology was proposed to evaluate the compliance of diverse server replicas. The methodology collects network traces to identify syntax and semantic violations, and to assist in their resolution. A tool called DiveInto was developed based on the methodology and was applied to three replication scenarios. The experiments demonstrated that DiveInto was capable of discovering various sorts of violations, including problems related with nondeterministic execution.

6.1.5 Proactive recovery

Individual replicas, even if fully patched, are still vulnerable to attacks that exploit flaws unknown to the security community (normally called *zero-day vulnerabilities*). Typically, these vulnerabilities are kept secret by the hackers until a suitable exploit is developed, and they are only discovered when eventually they start to be used to compromise systems. Although the number of zero-day vulnerabilities observed per year is relatively small, since they require specialized knowledge, effort and time to be found, there several of them that are discovered each year. Therefore, they cannot be disregarded in highly resilient solutions, and proper techniques should be employed to address them.

One however should keep in mind that the intrusion tolerant system remains correct as long as the adversary only controls up to f replicas. The problem occurs if a powerful attacker could silently discover flaws in $f + 1$ replicas, and then run the exploits at the same time taking over the system. *Proactive recovery (PR)* is a way to avoid this scenario [92, 98, 20, 73], by forcing replicas to be periodically rejuvenated with a diverse software configuration, which has a (hopefully) different set of vulnerabilities.

PR allows for: *i)* the cleaning of the faulty state in case the replica was silently compromised; *ii)* if a replica is correct but is being probed by an attacker, then a recovery will force the attacker to restart over because previously acquired knowledge is no longer of use. The system stays intrusion-tolerant as long as the recoveries occur faster than the $f + 1^{th}$ fault. Moreover, the recovery must modify the replica in such a way that it is not trivial for an attacker to compromise it again.

In previous works, the event that triggers the rejuvenation of a replica is based on time (e.g., every hour the replica is recovered). This can bring non trivial problems to the actual implementation of the PR mechanisms, since it needs to ensure that the trigger can not be delayed accidentally (e.g., due to heavy load in the system) or maliciously, and that the rejuvenation terminates within a bounded time [99]. To address this difficulty, we are developing within MASSIF other triggering actions, which at this point are based on an assessment of the risk level associated with a given replica in operation [50].

6.1.6 Reactive recovery

A limitation of proactive recovery is that a malicious replica can execute any action to disturb the system's normal operation (e.g., flood the network with arbitrary packets) until its recovery time, and there is little or nothing that a correct replica (that detects this abnormal behavior) can do to stop the faulty one. The observation is that a more complete solution should allow correct replicas to force the recovery of *detected or suspected* faulty replicas. This solution is called *Proactive-Reactive recovery* [98], and it can improve the overall performance of a system under attack by reducing significantly the amount of time a malicious replica has to disturb the normal operation.

If $f+1$ different replicas suspect and/or detect that replica R_j is failed, then this replica is recovered. This recovery can be done immediately, without endangering availability, in the presence of at least $f+1$ detections, given that in this case at least one correct replica detected that replica R_j is really faulty. Otherwise, if there are only $f+1$ suspicions, the replica may be correct and the recovery should be coordinated with the proactive recoveries in order to guarantee that a minimum number of correct replicas is always alive to ensure the progress of the system. The quorum of $f+1$ in terms of suspicions or detections is needed to prevent recoveries triggered by malicious replicas – at least one correct replica must detect/suspect a replica for some recovery action to be taken.

Notice that a proactive-reactive recovery service is completely orthogonal to the failure/intrusion detection strategy used by a system. The proposed service only exports operations to be called when a replica is detected/suspected to be faulty. In this sense, any approach for fault detection [13, 21, 44], system monitoring [40] and/or intrusion detection [41, 81] can be integrated in the system. For example, the observation of certain malicious actions performed by faulty replicas of the resilient core-MIS, such as altering the content of messages to be forwarded, can be used for this propose.

6.2 Resilient core-MIS architecture

A SIEM operates by collecting data from the monitored network and applications through a group of sensors, which then forwards the events towards a core facility for processing at a correlation engine (see Figure 6.2). The engine performs an analysis on the stream of events and generates alarms and other information for post-processing by other SIEM components. Examples of such components are an archival subsystem for the storage of data needed to support forensic investigations (RES in Chapter 8), or a communication subsystem to send alarms to the system administrators.

As SIEMs play an increasingly relevant role in the network and security management tasks of the organizations, it becomes imperative to ensure their correct operation under a wide range of fault scenarios. In particular, since security solutions often have been the target of malicious actions (e.g., anti-virus software [22], intrusion detection systems [7] or firewalls [61, 102, 23, 24]) as part of a wider scale attack, SIEM systems should be built/deployed under the assumption that this sort of actions will eventually occur. In this section, we will focus on the protection of the core facility components from outside attacks, improving the security and dependability of such essential processing.

The traditional solution to defend a critical network from malicious outsiders is a firewall. Firewalls are intended to separate security perimeters, such as a LAN from a WAN, and their main goal is to control the traffic that flows in and out of a facility. Typically, a firewall decides on letting a packet go through (or drop it) based on the analysis of its header and/or contents. Over the years, this analysis has been performed at different levels of the OSI stack, but the most sophisticated rules are based on

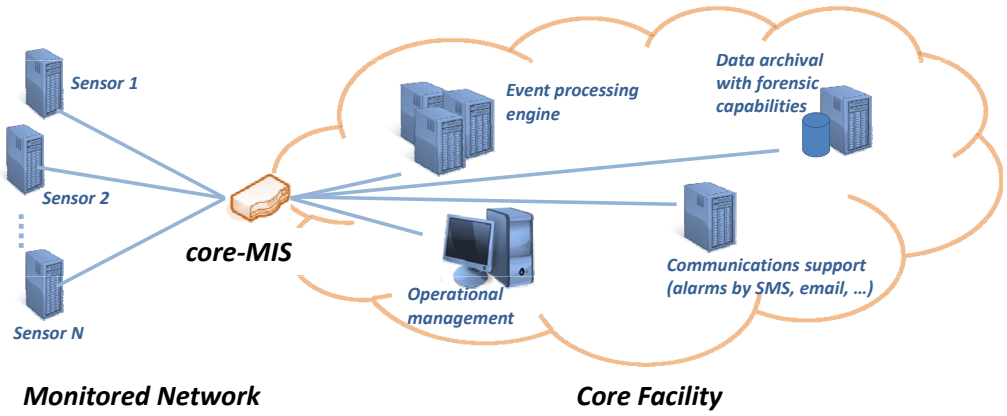

Figure 6.2: Overview of a SIEM architecture, showing some of the core facility subsystems.

the inspection of application data included in the packets. State-of-the-art solutions for application-level firewalls include network appliances provided by several vendors, such as Juniper[1], Palo Alto[2], and Dell SonicWall[3].

Firewalls are in general a single point of failure, and as such they have been the target of many attacks[4]. When successful, these attacks can impact on the system's security in different ways — for instance, they can allow complete access to the internal network resources or they can compromise availability by preventing traffic from going through the firewall. To address some of these issues, we propose a resilient core-MIS design that is able to remain operational under very harsh failure conditions, including both accidental crashes or the compromise of some of its components. The core-MIS will act as an application level firewall, in the sense that it will inspect the traffic from the outside networks towards the core facility, guaranteing that only good packets are allowed to proceed.

A key design choice on the core-MIS architecture is to divide filtering operations in two stages. The rationale for this decision was related with optimizing performance under attack and to give flexibility on the selection of the fault tolerance mechanisms. Performance is increased if common forms of attacks are handled earlier and with highly efficient tests. Therefore, the first stage, which is called *pre-filtering*, carries on lightweight checks on the messages and aims at discarding all attacks from external adversaries. In particular, it only lets messages go through that come from a pre-defined set of senders (typically, the edge-MIS) and that are correctly authenticated. Denial-of-service (DoS) traffic from external sources is immediately dropped, preventing these messages from overloading the next stage. The second stage is named *filtering*, and it is responsible for enforcing more refined application level policies, which can require the inspection of specific message fields and/or the observation of certain ordering rules (e.g., a sensor can only send data after some initial setup is performed with the engine).

Different fault tolerance mechanisms are employed at the two stages. Pre-filtering is implemented by a dynamic group components, whose size is adapted to the current demand. Hence, as the traffic load grows or extra edge-MIS have to be supported, the core-MIS creates more pre-filters to amplify

[1] http://www.juniper.net/us/en/products-services/security/
[2] http://www.paloaltonetworks.com/products/platforms/PA-5000_Series.html
[3] http://www.sonicwall.com/us/en/products/Next-Generation_Firewall.html
[4] An inspection of the Open Source Vulnerability Database (OSVD) shows that there have been many security issues in commonly used firewalls. During the 4 year period between 2009 and 2012, there were for example the following number of security reports: 36 for the Cisco Adaptive Security Appliance; 64 in Juniper Networks solutions; and 29 related to netfilter.

the aggregated processing capabilities (within the constraints of the hardware). Since pre-filters face the external network, they can experience various kinds of attacks and eventually be intruded. Therefore, we take the conservative approach of assuming that pre-filters can fail in an arbitrary (or Byzantine) way, meaning that they may for instance crash or start to act maliciously (when compromised by an adversary). The faults are tolerated by forcing the recovery of the erroneous pre-filters after their identification. The filtering stage is performed by a static group of filter components that are organized as a replicated state machine. Since we also want to consider filters that fail with an arbitrary behavior, the replication protocol needs to tolerate Byzantine faults and the final destination needs to vote on the results that are produced (to eliminate malicious data).

6.2.1 Design choices

Traditional firewall designs are based on a single logical component, and consequently, they are unable to tolerate most failures. A simple crash prevents the firewall from fulfilling its function, and more elaborated failure modes may allow an adversary to penetrate into the protected network. Some organizations deal with crashes (or DoS attacks) by replicating the firewall to support multiple entry points. This solution is helpful to address some (accidental) failures, but is incapable of dealing with an intrusion in the firewall. In this case, the adversary gains complete access to the internal network, allowing an escalation of the attack, which at that stage can only be stopped if other protection mechanisms are in place.

Over the past years, there has been a important amount of research in the development of systems that are intrusion-tolerant (see Section 6.1). To our knowledge, however, only very little work was devoted to design intrusion-tolerant protection devices, such as a firewalls (e.g., [98, 92]). Performance reasons might explain this, as Byzantine fault-tolerant (BFT) replication protocols are usually associated with reasonable overheads and limited scalability. For example, a straightforward implementation of a BFT firewall would require replicas to process every arriving message and then agree on their delivery. BFT solutions based on a leader can also become prey of an adversary, as they have a natural bottleneck replica that can be selected for the attack (instead of having to disperse the attack power over all replicas [5]).

The main motivation for this work is to address these limitations and propose a design for an intrusion-tolerant core-MIS that acts as a firewall. The fundamental design options that guided our solution were:

1. *Application-level filtering:* Allow for sophisticated filtering rules that take advantage of application knowledge. To support these rules, the core-MIS has to maintain state about the current communications. The state will have to be consistently replicated using a BFT protocol.

2. *Performance: a)* Address the most probable attack scenarios with highly efficient tests, and as earlier as possible in the filtering stages. *b)* Reduce communication costs with external senders, as these messages may have to travel over high latency links (e.g., do not require message multicasts).

3. *Resilience: a)* Tolerate a broad range of failure scenarios, including: malicious external attackers; compromised authenticated senders; and intrusions in a subset of the core-MIS components. *b)* Prevent malicious external traffic from reaching the internal network by requiring explicit message authentication.

6.2.2 System model

Since we want to address faults both of an accidental nature and also caused by a malicious adversary, we assume that erroneous components can behave arbitrarily (or in a Byzantine way). To be conservative, we assume that all failed components are controlled by a single entity that will make them act together to defeat the normal operation of the system. Therefore, they can for instance stop sending messages, skip some steps of the protocols, provide erroneous information to correct components, or try to delay the system.

We address three failure scenarios on the components, which provide increasingly more power to the adversary. The most common scenario occurs when an external adversary attempts to attack the internal network systems. He can deploy a large number of (unauthenticated) senders, whose aim is either to delay the communications or to bypass the core-MIS protection and reach the internal network. In particular, he can try to masquerade the messages as coming from normal senders, or perform a DoS by transmitting many erroneous packets to the core-MIS. However, as the core-MIS cannot stop DoS attacks that completely overload its incoming channels, which cause most of the normal messages to be dropped by the routers, it is assumed that the network includes other defense mechanisms to address this sort of attack (e.g., see [77]).

As the authenticated senders (e.g., the edge-MIS of the SIEM) are potentially spread over several outside networks, it is advisable to consider a second scenario where the adversary is also capable of taking control of some of these nodes. When this happens, it is assumed that the adversary gains access to all keys stored locally. Therefore, he will be able to generate traffic that is accepted by the core-MIS as correctly authenticated, but the packets will still be checked by the application level filtering. In any case, if the messages follow the rules, the core-MIS has to let them go through, as they are indistinguishable from other valid messages.

A third scenario occurs when the adversary is able to cause an intrusion in the core-MIS, and compromises a few of the pre-filters and/or filters. We assume that at most f_{pf} pre-filters fail of a total of $N_{pf} = f_{pf} + k$ (with $k > 1$), and that out of the $N_f = 3f_f + 1$ filters at most f_f fail. To enforce this assumption it is necessary to ensure that core-MIS components fail independently, which typically can be achieved with good coverage if one employs diversity [49]. A failed pre-filter can for instance modify the received traffic or generate invalid messages that are given to the filtering stage. A malicious filter can perform similar attacks, and in addition send erroneous data to the final destination.

We assume that the communications with the core-MIS can also suffer from accidental faults and/or attacks. Thus, packets may be lost, delayed, reordered or corrupted. In the MASSIF resilient architecture, most of these faults will be tolerated by the REB, as it improves the robustness of the communications. However, we might envision deployment scenarios where the core-MIS might be used without the REB support. In this case, the some of faults can be tolerated with the help of SIEM applications, for example, by retransmitting the packets. To make the rest of description as general as possible, we will present the core-MIS protocols assuming that a very simple communication support exists to connect the edge-MIS and the core-MIS, namely we will consider that UDP is employed[5].

[5]If the REB is used for the communication then we can remove the authentication check between the sender and the pre-filter, but the rest of the protocol basically remains the same.

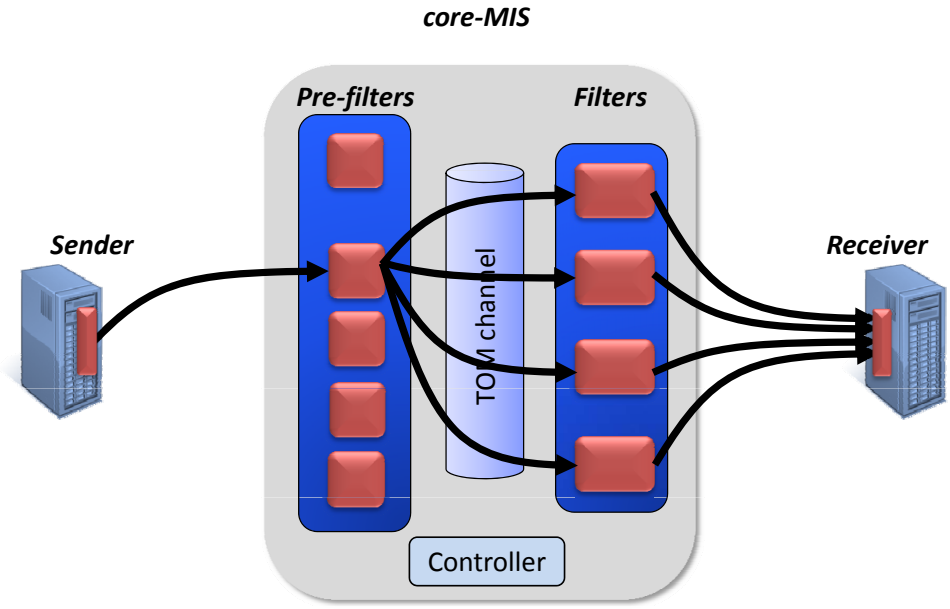

Figure 6.3: Overview of the architecture and communication pattern with the core-MIS.

6.2.3 Core-MIS operation

This section describes the core-MIS execution when messages arrive from the senders and are forwarded to the final destination. Traffic identified as malicious by one of the filtering stages is discarded. Figure 6.3 displays the architecture of the core-MIS and shows the communication pattern among the components.

Components of the architecture

The main components involved in the operation of the core-MIS are:

Sender-module There is a communication module deployed on the sender's side that is responsible for the interactions with the core-MIS. It performs the authentication with the core-MIS, and encapsulates the data received from the application to ensure its correct delivery. Each sender-module is associated with one preferred pre-filter that is selected during setup. Messages are transmitted towards the preferred pre-filter.

Pre-filters They appear as the external interface of the core-MIS, and perform basic filtering actions. Although tests are kept simple to improve efficiency, they are nevertheless effective at deterring most attempts of DoS. Pre-filters forward to the filters the accepted messages using a *Byzantine total ordered multicast (TOM)* protocol [97]. This protocol ensures that all filters receive identical messages in the same order.

Filters They implement a state machine replication service that filters messages based on application knowledge. Each of them acts as a replica, applying exactly the same rules to every message that was deemed valid by the pre-filters. Therefore, correct filters should reach to the same conclusion regarding the validity of the messages. Accepted messages are transmitted to the final destination.

Receiver-module It is a communication module deployed on the receiver's side, whose responsibility is to deliver the messages to the SIEM application. The main role of this component is to vote Filters' messages in order to accept only correct messages (recall that compromised filters may send invalid data). A message is considered correct is $f_f + 1$ filters vouch for it.

Controller Is a trusted component of the core-MIS that runs with a higher privilege level (depending on the actual implementation, it can be developed in different ways; see Section 6.2.4 for a discussion). The controller takes input from the filters to decide on the creation of more pre-filters, or to delete one of them.

The deployment of the core-MIS requires a *key distribution scheme* to create shared keys between sender-modules and the pre-filters, and between filters and receiver-modules. These shared keys can be distributed based on some long term secrets (e.g., private-public key pairs), using for instance protocols similar to IPsec (the Internet Key Exchange [58]).

Transmitting a message

The sender-module receives from the application a buffer with $DATA$ to be transmitted to a certain destination behind the core-MIS. The buffer needs to be encapsulated in a message with some extra information required to protect the communication. It is necessary to add the sender-module identifier c_i and a sequence number value sn. The sequence number is incremented in every message, and is used to prevent certain replay attacks either from the network or a compromised pre-filter.

Next, information is added to the message to protect its integrity and allow for its authentication. A signature $sign$ is performed over the messages contents, and a MAC M_{pf} is obtained for the pre-filter associated with this sender (covering all fields, including the $sign$). The MAC is computed using a shared key established with the pre-filter, and serves as an optimization to speedup checks (as testing a MAC is faster than a signature). The signature is calculated with the private key of the sender-module (the corresponding public key is provided to the core-MIS components for signature verification, during the setup of the system). The message then is sent via UDP to the pre-filter pf:

$$M = \langle c_i, sn, DATA, \text{sign} \rangle_{M_{pf}}$$

Pre-filter Upon receiving the message, the pre-filter applies a few checks to determine if the message should be forwarded or discarded (see Figure 6.4):

(a) White list: each pre-filter maintains a list with the nodes that are allowed to transmit messages (i.e., which were authorized by the system administrator). Messages coming from other nodes are simply dropped. This check is based on the address of the sender of the message.

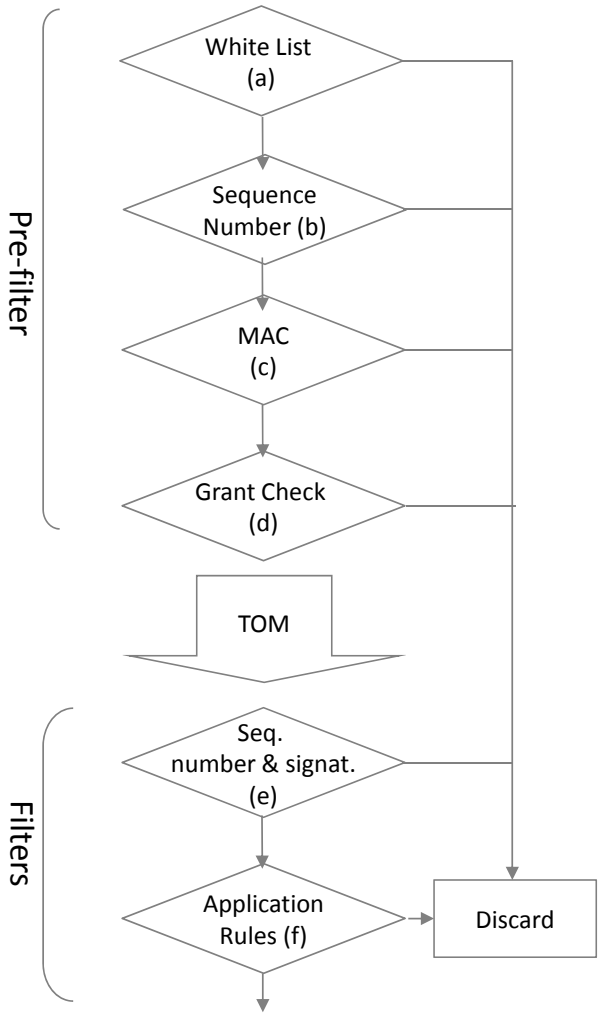

Figure 6.4: Filtering stages at the core-MIS.

(b) Sequence number: determines if a message with sequence number sn from sender c_i was already seen. In the affirmative case, the message is discarded to prevent replay attacks. Furthermore, messages are also dropped if their sn is much higher than the largest sequence number ever seen from that sender (a moving window of acceptable sequence numbers is used).

(c) MAC test: MAC M_{pf} is verified to authenticate the message contents. If the check is invalid, the message is dropped and the sequence number information is updated to indicate that this message was not seen.

(d) Grant check: each pre-filter controls the amount of messages that sender-module is transmitting. Messages that fall outside the allocated amount are dropped, to ensure that all senders get a fair share of the available bandwidth (and to avoid DoS attacks by intruded senders).

Then, the pre-filter invokes the total order multicast channel to forward the message to the filters. This channel ensures that all filters receive the message in the same order.

Filter Each filter applies identical checks to the message:

(e) Sequence number and signature test: as the pre-filter might have been intruded, it is necessary to perform an extra check on the integrity/authenticity of the message fields. The test on the sequence number is similar to the pre-filter, but then the signature is used to ensure that all filters reach the same decision regarding the validity of the contents.

(f) Application-level rules: apply the application-level filtering rules to determine if the message is acceptable. These rules may look into the contents of $DATA$ and relate it with context information about messages that were previously processed, i.e., we could support iptable rules.

Although uncommon in several deployment scenarios, it can happen that the network re-orders the delivery of messages. The impact of this problem is that application-level rules may drop some of the out of order messages, which later on will have to be re-transmitted by the SIEM application. For example, if messages A and B should appear in this sequence but are re-ordered, then the rules may consider B invalid and then accept message A. At some point, the application would determine that B was lost, and would re-send it.

To address this issue, each filter enqueues for a while messages with a sequence number greater than the expected (but that do not exceed a threshold above the last processed sequence number). Next, it will continue to process other messages, until either: 1) the missing message(s) arrive, and then they are all tested in order, or 2) it gives up on waiting, and checks the message. This last decision is made after processing a pre-determined number of other messages.

Messages that are considered valid are encapsulated in a new format, and are then sent to the final destination. Basically, the filter removes the signature and substitutes the MAC with a new one M_f. This MAC is created using a shared key between the filter and the receiver-module.

$$M1 = \langle c_i, sn, DATA \rangle_{M_f}$$

The receiver-module accumulates the messages that arrive from the various filters, until a quorum is collected to ensure correctness. Furthermore, it validates the MACs to authenticate the content. A message can be delivered to the application when there is a quorum of $f_f + 1$ equal copies, as this indicates that at least one correct filter accepted the message.

Handling component failures

In the Byzantine model, every failed component can behave in an arbitrary way, intentionally or accidentally. Therefore, when designing the core-MIS, it is necessary to incorporate mechanisms that are resilient to very harsh failure scenarios. Given the architecture of Figure 6.3, one has to address faults in the pre-filters and filters, and needs defences against erroneous (authenticated) senders and external attackers (regarding the receiver-module, there is not much that can be done about its failures).

Although pre-filters carry out the same function, i.e., check the messages arriving from a few sender-modules, they are not replicated. Furthermore, since pre-filters face the external core-MIS interface, there is a higher risk of being compromised. As such, in order to keep the core-MIS operational, it is required that failed pre-filters are identified and recovered. We leverage from the filter setup to perform

the fault detection (of either crashes or misbehavior), and then use the controller to re-start erroneous pre-filters.

Filters implement state machine replication, and as long as they process the same messages in identical order, identical results should be produced. Consequently, filter faults can be tolerated by employing a voting technique that selects results supported by a sufficiently large quorum (as explained above, an output with at least $f_f + 1$ votes).

Pre-Filter failures Since pre-filters can fail arbitrarily, they can exhibit very different invalid behaviors. Moreover, sometimes they may look as acting in a flawed way, but in fact they are correct. For example, when a pre-filter is under a DoS attack, messages can start to be lost on the network due to buffer overflows, but this is indistinguishable from a failure that causes omissions. This sort of ambiguity precludes exact failure detection, and therefore, our aim should be to provide a number of mechanisms that allow the core-MIS to recover from failures and continue to deliver a correct service (i.e., let clients send messages to a given destination). One however should accept right from the start that occasionally there might be mistakes on the fault detection — a good pre-filter may be erroneously considered failed (e.g., it is just overloaded), while a flawed one might go undetected (e.g, if messages are only dropped rarely).

An initial step on the detection process is for the pre-filter to evaluate its own conduct. In particular, it observes the amount of traffic that is arriving to determine if there is a risk of becoming overloaded. An easy way to carry out the analysis is to measure the waiting intervals for message arrivals over a certain period. If those intervals are very small on average, then the per-filter is working at its full capacity or there is already some overload. When this happens, the pre-filter sends over the TOM channel a $WARNREQ$ request to the filters, so that they may take some action to solve the problem (see below).

To detect failures in general, it is necessary the cooperation of the filters and the senders. Since a sender generates the data that is transmitted to a pre-filter, it knows how many messages should have arrived to the filters. Consequently, by providing a mechanism where a sender can tell the filters how many messages were supposed to be delivered, it is possible to determine if many omissions are occurring on a pre-filter (or in the network connecting to it). The procedure is the following:

- Periodically, the sender-module transmits to the filters a special $ACKREQ$ request, where it indicates the sequence number of the last message that was transmitted (plus a signature and a MAC to ensure authenticity). This request is first sent to the preferred pre-filter, but if no answer is received within some time, it is forwarded over the other pre-filters. The waiting period is adjusted in each retransmission by doubling its value.

- When the filters get the request, they use the included sequence number together with some local information, to find out how many messages have been missing. The local information is basically the set of sequence numbers of the messages that were correctly delivered since the last $ACKREQ$.

- Based on the number of missing messages, the filters transmit through the same pre-filter a response $ACKRES$ to the sender-module, where they state the observed failure rate and other control information (plus a signature). Filters may also specify some recovery action if the failure rate is too high (see below).

Finally, filters can also learn about failures based on the validations performed on the messages. For example, an invalid signature check is highly suspicious because all messages corruptions should be

captured by the pre-filters with the MAC test. This would indicate that either the pre-filter or/and the sender is failed (since it could have generated a wrong signature but a good MAC). Filters may also become suspicious if there is a sudden increase on the level of out-of-order message arrivals. This could indicate an attack on the network or a malicious pre-filter. The filters should attempt to fix these behaviors when they are observed.

Three kinds of recovery actions are used to solve the above mentioned problems. These actions are taken depending on the extent of the perceived failures, but they should be safe from the point of view of the core-MIS operation:

- Redistribute load: For example, if a pre-filter provides a warning about its load, or high failure rates start to be observed for a specific pre-filter, the first course of action is to move some of the message flows to other pre-filters. This is achieved by specifying in the $ACKRES$ response of a sender-module the identifier of a new pre-filter that should be used. At that point, the sender-module is expected to contact the chosen pre-filter to initiate communications through it.

- Increase pre-filtering capacity: If the existing pre-filters are unable to process the current load, a second course of action is to create more pre-filters (until a certain maximum is reached, depending on the hardware resources). To accomplish this, the filters contact the controller informing that a extra pre-filter should be started. When the controller receives $f_f + 1$ of such messages, it performs the necessary steps to fulfill the request (which are dependent on the actual deployment of the core-MIS). The new pre-filter begins by doing a few startup operations, which include the creation of a communication endpoint, and then it uses the TOM channel to inform the filters that it is ready to accept messages from the sender-modules.

- Kill pre-filters: When there is a reasonable level of suspicion on a pre-filter, the safest course of action is to have the filters ask the controller to destroy it. Moreover, if the load on the core-MIS is perceived as having decreased substantially, the filters select the oldest pre-filters for elimination, allowing eventual aging problems to be addressed. The controller carries out the needed actions when it gets $f_f + 1$ of such requests (once again, these depend on how the core-MIS was deployed). The sender-modules that may have had their flows affected, will be informed about the replacement pre-filter through the $ACKREQ$ mechanism (they will use another pre-filter to send the request, and get the information about a new pre-filter to be used in the response).

Filter failures Since filters receive the same messages and execute in a deterministic way, they are expected to produce the same results. Therefore, it is possible to detect erroneous behaviors by comparing the outputs of the filters. Namely, when a receiver-module sees divergent messages being forwarded, for a specific sequence number of a given sender, this provides evidence that filters may have failed. Additionally, the controller should receive similar requests to update the pre-filters configuration, and missing or disagreeing messages also give an indication of a problem. Since we anticipate that filter failures will be rare, we decided to exclude automatic filter recovery — the component that discovers a misbehaving filter sends a warning to the system administrator, so that recovery can be initiated manually.

Sender-module failures By looking at the arriving messages, the filters may also detect some sender-module failures. In particular, we are concerned with behaviors that may influence the normal execution of the core-MIS. For example, if a correctly signed message arrives with a much larger than expected sequence number, this gives a clue that something may be wrong with the sender. More serious is a

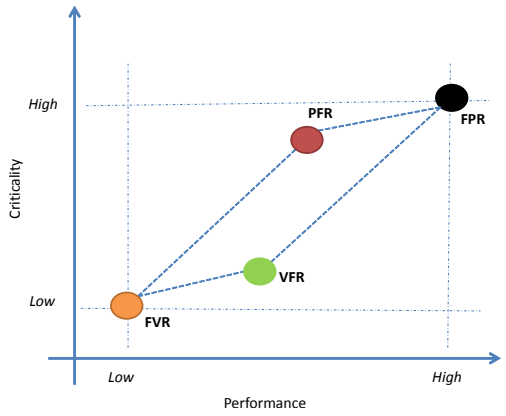

Figure 6.5: Tradeoffs in some deployment alternatives.

sender-module that is constantly complaining about the pre-filters (e.g., by apparently showing high failure rates), or that is transmitting at a speed above the allowed by the grant check (test (d) of Figure 6.4). Here, some defense action needs to be carried out, as these cause the core-MIS to spend extra resources (e.g., constantly moving the sender through the various pre-filters, or wasting network bandwidth).

To be conservative, we decided to follow a simple procedure to protect the core-MIS from a specific sender-module:

Filters maintain a counter per sender that is increased whenever new evidence of failure can be attributed to it. When the counter reaches a pre-defined value, the sender is disallowed from communicating with the core-MIS by temporarily removing it from the white list (test (a) of Figure 6.4) and a warning is sent to the system administrator. To let excluded senders regain access to the service, the counter is periodically decreased. When the counter falls below a certain threshold, the sender is moved back into the white list.

6.2.4 Overview of deployment alternatives

The core-MIS requires several separate components (filters and pre-filters) for effective deployment. However, costs are a major concern of any administrator. Therefore, we present a few deployment alternatives that can be made based on our solution. In any case, one must keep in mind that resilience usually has associated costs.

Figure 6.5 presents the rational for making deployment decisions. The considered solutions try to tradeoff costs with the use of virtualization [88, 111, 107]. In all options, different components run in separate virtual machines and/or physical machines (or hardware boards). A solution with more physical machines is desirable for more critical systems, due to the higher fault isolation and also because of the potentially better performance. With virtualization, each physical machine supports several virtual machines, which means that there might be less machinery costs but performance can be reduced as resources are shared. Although virtual machines provide some level of isolation among the different components, preventing in most cases intrusions from propagating from one replica to the others, hardware faults may have an impact in all replicas.

- Full Physical Replication (FPR): every pre-filter and filter runs in a different physical machine.

- Full Virtual Replication (FVR): every pre-filter and filter runs in different virtual machines in the same physical hardware.
- Virtual Filter Replication (VFR): pre-filters and filters run in virtual machines, but the pre-filters are in a physical machine and the filters are in another physical machine.
- Physical Filter Replication (PFR): pre-filters run in virtual machines of the same physical machine, and each filter runs in a separate physical machine.

The controller should be run in a separate component that is protected from failures in the rest of the system. In the case of a virtualized environment, it can be implemented in the hypervisor, allowing complete access to the other components. With physical replication, there are a few alternatives, such as implementing it in a separate hardware module that can force the rebooting of a machine, or hybrid solutions that use a separate control network and a privileged software module [98].

6.3 Experimental evaluation

This section presents an experimental evaluation of the current implementation of the core-MIS. Previously, we have shown in D2.3.3 - Acquisition and Evaluation of the Results [31] some of the security properties of the core-MIS, such as integrity and authentication of messages. These properties are guaranteed using the secret key shared by the sender and the pre-filter, allowing them to add (and check) a MAC to the transmitted messages. It supports the discovery of several kinds of attacks, such as the corruption and creation of invalid messages. In this document, we focus the analysis on the performance of the core-MIS under high workloads. Our tests emulate the event traffic of the Olympic Games, as the payload of the messages exchanged in the benchmark was created based on the event data provided by this scenario. The analysis also considers some aspects of the core-MIS' resilience, namely the performance under a denial-of-service (DoS) attack.

6.3.1 Implementation and testbed

The experimental testbed was composed by several components, some of them emulate real entities such as the sender and the attackers, and the remaining components are part of the core-MIS implementation. The protocol follows approximately the description of Section 6.2, with a few changes mentioned below:

Sender. In order to emulate the edge-nodes, we implemented a communication module called *sender*. This component communicates with the core-MIS via UDP. Since this is a core-MIS evaluation, we did not use the REB to support the communication between the edge and the core-MIS node. To increase the robustness of the system, the edge node sends the packets to $f + 1$ pre-filters, where f is the number of tolerated faulty pre-filters. In all experiments, the value of f was set to 1.

Pre-filters. They stay between the sender node and the filters. Upon receiving a message, the pre-filters verify if the sender belongs to the authorized list, and then forward the accepted messages to the filters. In order to ensure that all the filters receive the same messages in the same order, the communication is made with a *Byzantine total ordered multicast* protocol, implemented by the BFT-SMaRt library [97].

Filters. They are the most complex part of the core-MIS, as they are implemented as a state machine replication service. Each filter acts as a replica, applying exactly the same rules to every message received from the pre-filters. The filter's first verification is to determine if the message's sequence number is valid. Each sender shares a secret key with each filter to authenticate the sender and verify the message's integrity with a MAC. The accepted messages are transmitted to the final destination, i.e., the post-filter.

Post-filter. It is a back-end module, whose responsibility is to deliver messages to the SIEM application. However, we did not use any SIEM engine to correlate the events in order to simplify the experiments. The main role of this component is to vote the filters' messages in order to accept only correct messages (recall that compromised filters may send invalid data). A message is considered correct if $2f_f + 1$ filters vouch for it.

Attacker(s). One of the goals of these experiments is to test the core-MIS' resilience under a DoS attack. Therefore, to emulate an attacker behavior, we developed a simple program that sends small packets (1 byte) via UDP at full rate transmission to overload a pre-filter forwarding capacity. These attacks target only one of the pre-filters, on the same network port used by the sender to communicate.

The testbed included six physical machines in a local network. The machines have two Quad-core Intel Xeon 2.27 GHz CPUs, with 32 GB of memory, and a Broadcom NetXtreme II Gigabit network card. In order to emulate the DoS attacker, we used three machines from another cluster. Each one of these machines has a Dual-core Intel Pentium 4 CPU 2.80 GHz, with 2 GB of memory, and a Broadcom NetXtreme Gigabit network card. The network was setup over a 1 Gbit/sec HP E3500yl-24G-PoE+ switch.

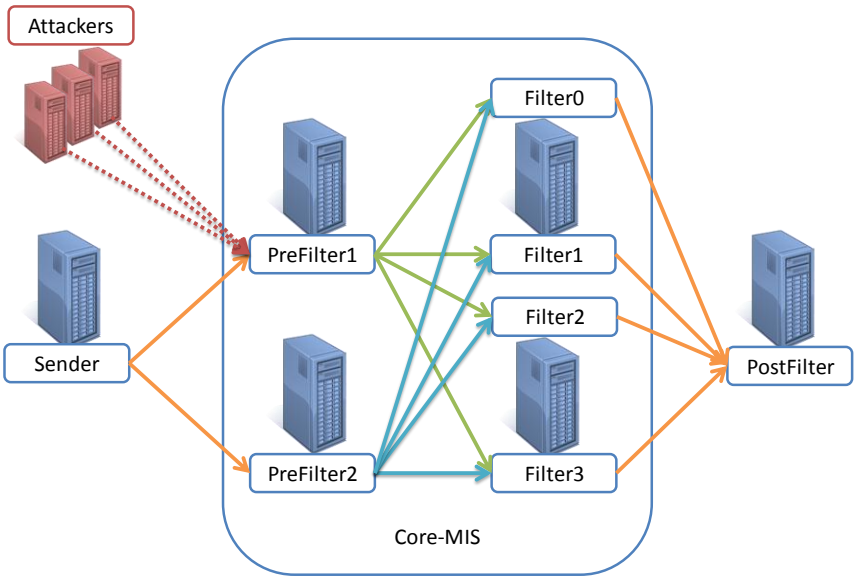

Figure 6.6: Core-MIS testbed architecture used in the experiments.

Figure 6.6 shows the testbed architecture, which is based on the Full Physical Replication configuration (see Figure 6.5). The pre-filters and filters execute in different machines, but the four filters instead of running in four machines are divided in two machines (due to the limited number of resources). Each machine runs the Ubuntu 10.04 64-bit server edition (LTS version). Every core-MIS component was developed in Java 7 programming language, and the *attacker* software was developed in Python programming language.

6.3.2 Benchmark description

The experiments were performed using two main workloads. The first one assumes that the edge node sends one event per packet. The event size is 120 bytes, which is the average of the Olympic Games events based on the log analysis. The second one considers that the edge node does some event aggregation. The events are aggregated in groups of seven events, resulting on 840 bytes of payload size. To this data, the sender then adds information to identify the parties and to protect the communications from the attacks (e.g., the MACs).

The experiments were setup with a sender that transmits at a reasonably high rate, in the order of 7000 messages per second. The throughput measurements were made in the post-filter component, where it counts the number of messages delivered per second. The post-filter receives $3f + 1$ copies of a message from the filters and then performs a voting operation on them. Therefore, in each second, we count the number of messages that are accepted after voting (i.e., that are delivered to the application).

For each workload, we analyzed the system's performance in the normal case and also under a DoS attack. In both workload experiments, the attack takes around 200 seconds (which can be observed in Figures 6.8 and 6.10).

We also performed a latency evaluation for different packet sizes, from 120 to 960 bytes. This evaluation was carried out in three steps. First, we measured the round trip time (RTT) to transmit a message through the core-MIS. To achieve this, the sender transmits one packet through the core-MIS towards the receiver (i.e., the post-filter). Next, the receiver uses an alternative path, which does not go through the core-MIS, to return an acknowledgement. The sender calculates the elapsed time between the two events, i.e., message transmission and the reception of the acknowledge. Second, we measured the latency of transmitting an acknowledgement (LACK) over the alternative path. Third, the latency of the communication over the core-MIS is computed by subtracting LACK from RTT (i.e., latency = RTT - LACK). Each measurement was repeated a large number of times to get confidence on the results.

6.3.3 Performance results

In the normal case the system sustains a throughput of around 7000 messages per second, as can be observed in Figure 6.7. As expected, there is some variation on the exact number of messages that is delivered in each second, as this value depends on how the messages are enqueued and batched among the various components of the core-MIS. The core-MIS architecture has several layers between the sender and the post-filter, i.e., the final receiver, and each layer adds some entropy. Every pre-filter and filter has a few message queues in order to handle more requests. Sometimes, when a component consumes messages from the queue, there is not enough messages to reach the highest system's throughput. Additionally, the total order multicast channel utilizes several optimizations, and among them it employs a batching technique where a group of messages is ordered and delivered simultaneously.

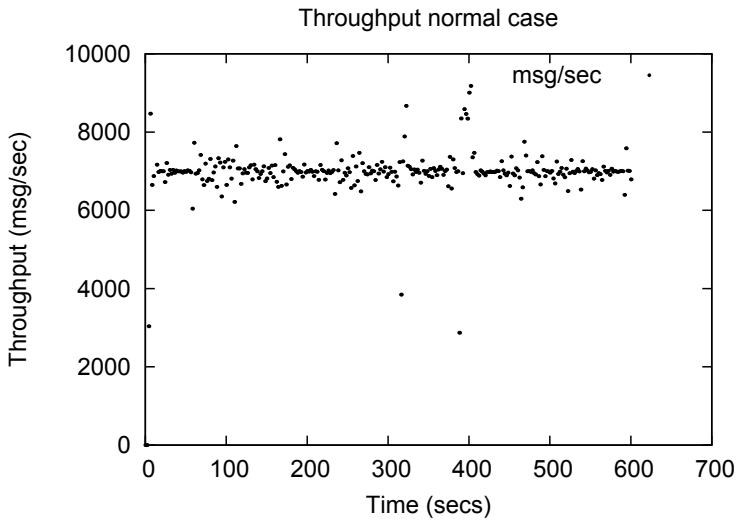

Figure 6.7: Core-MIS throughput (with a single Olympic Game event per packet).

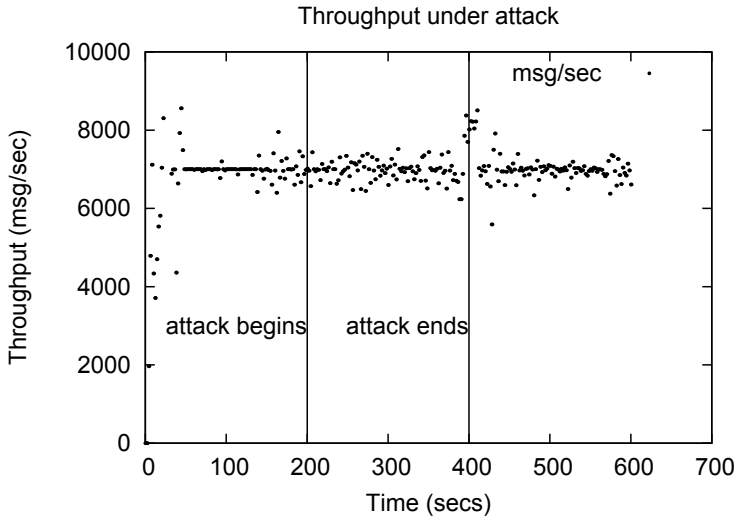

Figure 6.8: Core-MIS throughput during a DoS attack (with a single Olympic Game event per packet).

In any case, the experiment demonstrates that the core-MIS is able to process a large number of messages, which is more than sufficient to accommodate the loads envisioned in the Olympic Games scenario (at a throughput of 7k messages per second, the core-MIS is able to deliver more than 600 million events per day; this is more than an order of magnitude more events than what is required for this scenario).

Figure 6.8 displays the behavior of the system when a DoS attack is carried out, starting at second 200 and with a duration of 200 seconds. It is possible to observe that throughput of the core-MIS does not suffer a significant impact during the attack. There is only a sightly higher variance on the throughput computed at each second, but the system stabilizes again when the attack is terminated. This behavior is explained with the replication of the pre-filter and the tests that are performed to drop malicious traffic

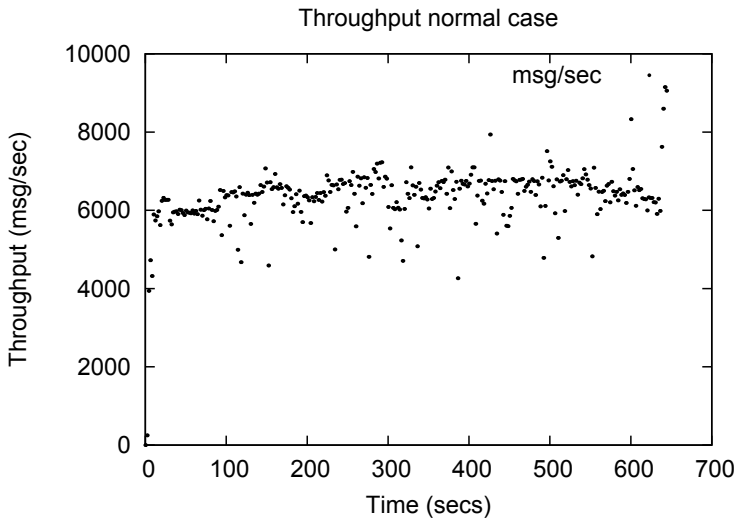

Figure 6.9: Core-MIS throughput (with several Olympic Game events aggregated per packet).

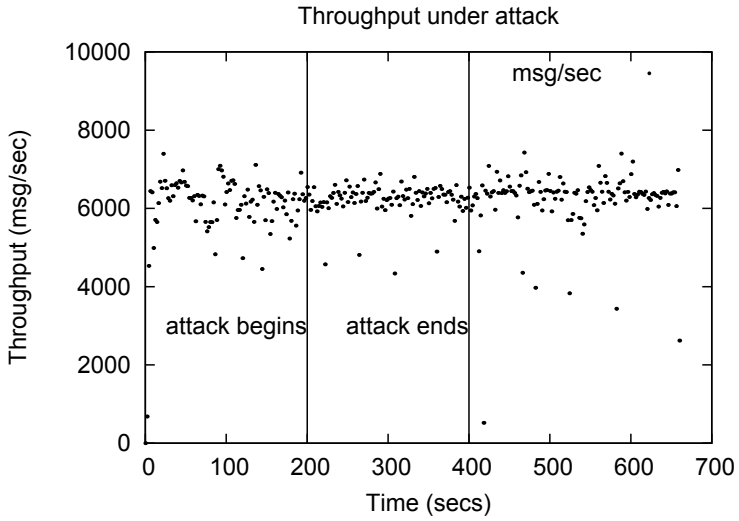

Figure 6.10: Core-MIS throughput during a DoS attack (with several Olympic Game events aggregated per packet).

as earlier as it is feasible. Since the sender transmits the packets to both pre-filters, upon the attack, the redundant pre-filter continues to deliver the packets.

In the next experiment, we studied the performance of the core-MIS in a setting where aggregation is done at the edge-MIS. Figure 6.9 shows the throughput when 7 events are placed in every message. From the graph, it possible to see that at the beginning (until the first 100 sec), the core-MIS delivers around 6000 messages per second. Later on, the value increases, reaching approximately the same throughput as observed previously. These small differences have to do with filling up of the various queues during an initial period. Even so, the experiment demonstrates that small levels of aggregation does not impact the core-MIS message throughput, and therefore, is quite beneficial as many more events can be delivered

per day (at a rate of 7K messages per second, it is possible to deliver more than 4 billion events per day; this is more than two orders of magnitude more than what is required by the olympic games scenario).

Figure 6.10 displays how the system behaves under a 200 seconds attack. The graph shows no significant difference between the periods under attack and in the normal conditions. This shows that the architecture is able to address this attack without compromising its normal functioning.

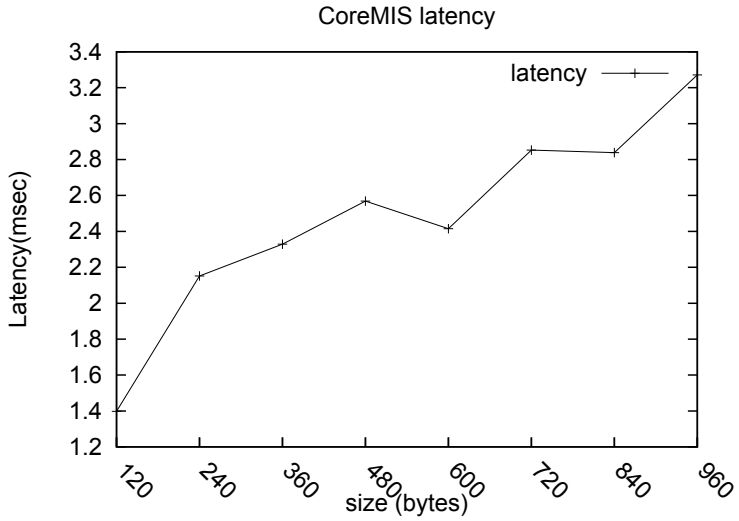

Figure 6.11: Core-MIS average latency for messages with distinct payload sizes.

The latency imposed by transmitting a message through the core-MIS is presented in Figure 6.11. Values are presented for increasing message sizes, as more events are aggregated to the payload (recall that the average event size is 120 bytes), until IP fragmentation starts to be needed. As expected, one can observe that when the message size grows the latency also increases. This occurs not only because larger messages take longer to be transmitted through the network, but also because cryptographic operations require more processing when data is bigger (e.g., the validation of a MAC involves all bytes of the message to be checked). In any case, latency values are in the order of a few milliseconds, which shows that the core-MIS does not impose delays that would prevent events from being correlated at the SIEM engine in near real-time.

7 Resilient Complex Event Processing

In this chapter, we discuss the Complex Event Processing fault tolerance protocol. In distributed environments, the need for fault tolerance comes from the observation that the higher the number of nodes, the higher the probability of experiencing faults. Existing fault tolerance protocols have not addressed aspects such parallel execution of operators, dynamic load balancing and elasticity, which are the innovative aspects of the MASSIF CEP as described in [27], [29] and [30]. Our fault tolerance protocols must comply with the following requirements:

1. **Low runtime overhead and fast recovery**. The most important requisite of any fault tolerance protocol for a stream processing engine is to provide low runtime overhead and fast recovery. Runtime overhead must be kept as lower as possible as the protocol is useful as long as its impact on the normal processing (e.g., its impact on the result latency) does not violate the application requirements. On the other hand, recovery time should be as short as possible in order to reduce the quality loss and the user satisfaction upon failures. We stress that, among the two requirements, low runtime overhead is usually more important than recovery time as failures are considered to happen occasionally. That is, the runtime overhead is a price that is being paid continuously while recovery time is a price paid sporadically.

2. **Precise Recovery**. We look for a fault tolerance protocol that guarantees precise recovery because most of the applications do not allow for data loss.

3. **No replicas**. Although replicas can lead to fast recovery times, its overhead in terms of computation and resources is too high. In an environment as the cloud, where the number of nodes is usually optimized to reduce costs, duplicate (or triplicate) the system size to maintain replicas is not a feasible solution. As an example, a system that requires two replicas for each primary node will result in an overhead of 200%.

4. **Not rely uniquely on main memory**.Streams are defined as potentially unbounded sequences of tuples. Thus, solutions that limit the scope of a stream are mandatory when relying on main memory (e.g., the windowing mechanism). When discussing a fault tolerance protocol that does not rely on replicas, we cannot make the assumption that all the states maintained by all the instances running a set of queries can be aggregated using the main memory of a small number of dedicated servers. For this reason, we look for a solution that leverages the use of persistent storage.

5. **Decouple state maintenance from topology**. As discussed in [30], elasticity and load balancing techniques are based on state transfer protocols used to move part of the state of an operator to another one. Due to the fact that the state of an operator is not statically assigned to it but it can rather be partitioned and transferred at runtime, the fault tolerance protocol will need to be designed decoupling state maintenance from any particular topology. That is, the protocol will

not maintain a copy of the state of a particular physical node, it will rather maintain a copy of the whole state of the parallel-operator independently of the number of machines being used to process it at any point in time.

6. **Tolerate failures happening during reconfigurations of the system**. The last requirement of the fault tolerance protocol is that it must tolerate failures happening while the system is being reconfigured (i.e., failures happening while the state of an operator is being transferred or while nodes are being provisioned or decommissioned).

7.1 Intuition about fault tolerance protocol

In this section, we provide an intuition about how fault tolerance is provided to data streaming operators. Without focusing on how to implement such technique, let us first consider that we have a means for maintaining all the tuples that are being exchanged between two nodes A and B. In case of failure of node B, we can recreate the lost state replaying the past tuples to a replacement instance $B0$. Nevertheless, the following considerations hold:

1. In order to recreate the same state, tuples should be reprocessed in the same order they were processed by the failed instance.
2. In order to reduce the recovery time, the tuples that should be forwarded again should be only the ones that contributed to the state of node B just before its failure.

The simplest solution could be to persist to disk all the tuples forwarded by the upstream node A and, upon failure of B, to replay all the stored tuples to $B0$. This solution has several issues. First of all, the impossibility of maintaining all the past history of a stream (streams are potentially unbounded sequences of tuples) on the disk. Furthermore, even imposing the limitation on the amount of tuples that must be maintained, the runtime overhead to replay all the stored tuples would be too high. Another aspect to consider is related to the peer nodes upon which the node we want to protect from failures relies. A node relies on its upstream peers in order to maintain past tuples (it cannot rely on itself otherwise it will lose them upon failure). Consider now a scenario where the tuples being forwarded between the node we want to protect from faults and its downstream peer are being extremely delayed due to a congestion in the channel (e.g., the tuples produced minutes ago have not yet been received by the downstream node). Upon the failure of a node, if we just recreate the latest state maintained by the node, part of the tuples being forwarded that were not received by the downstream node might get lost. For this reason, the node relies on its downstream peers in order to make sure that, each time we update the point in time from where to replay tuples, all the previous output tuples generated by the node have been received. Finally, the last aspect to take into account is related to the information maintained in a generic stateful operator state. While data is being processed, new windows are created and existing windows are slid. At the same time, windows might become obsolete and should be garbage collected to reduce the state that must be recovered upon a failure. To give an idea about how an operator can be recovered in case of failure, let us consider the case of an aggregate operator used to compute the average value on the field X of the incoming tuples. The operator is defined with a time-based window with size and advance of 180 and 120 seconds, respectively [29]. That is, every 2 minutes, the operator outputs the average value on X of the tuples received during the last 3 minutes.

As presented in Figure 7.1, assume the aggregate operator (Agg) is deployed at node B, its input tuples are forwarded by an operator deployed at node A, while its output tuples are being sent to the

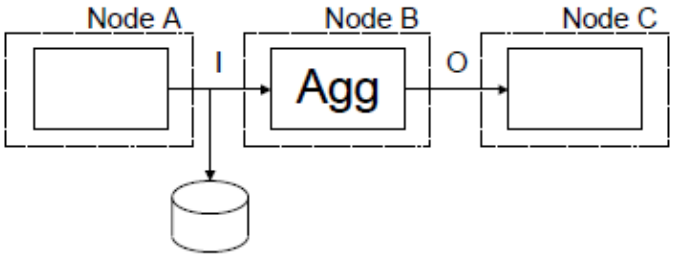

Figure 7.1: Example of fault tolerance for aggregate operator.

Sample input and output tuples for operator B							
IN		t1	t2	t3	t4	t5	t6
	ti.Time(secs)	0	60	120	180	240	300
	ti.X	7.8	8.2	8	7.5	7.3	8.1
OUT		-	-	-	T1	-	T2
	Ti.Time(secs)	0	60	120	180	240	300
	Ti.avg(X)	-	-	-	8	-	7.6

Table 7.1: Input and output tuples

operator running at node C. Assume also that tuples exchanged between nodes A and B are being persisted and, at any point time, stream I can be replayed starting from a given tuple. Table 7.1 presents the sequence of tuples consumed and produced by the operator.

Input tuples are denoted as ti while output tuples are denoted as Ti. For the ease of the explanation, we only consider fields $Time$ and X for the operator input tuples. The first window managed by Agg contains the tuples with $Time \in [0; 180[$; in the example, tuples $t1$; $t2$; $t3$. The second window managed by Agg contains the tuples with $Time \in [120; 300[$; in the example, tuples $t3$; $t4$; $t5$, and so on. Tuple $T1$ is produced upon reception of tuple $t4$ as the latter causes window $[0; 180[$ to slide ($t4.Time$ falls outside the window). Tuple $T1$ carries the average X computed from tuples $t1$; $t2$; $t3$. After sliding the window, tuples $t1$; $t2$ are discarded. Similarly, tuple $T2$ is produced upon reception of tuple $t6$. Consider how Agg state changes before and after processing tuple $t4$. Before processing $t4$, three tuples are currently maintained by A (i.e., tuples $t1$; $t2$; $t3$). After processing $t4$, the new state includes tuples $t3$; $t4$. Imagine that the instance running operator Agg fails just after outputting tuple $T1$. If tuple $T1$ has been received by node C, the node replacing B can be fed starting from tuple $t3$. This way, the first tuple the replacement instance will produce is $T2$. Doing this, from the point of view of the operator deployed at node C the failure has been completed masked. If tuple T1 has not been received by node C, the failure will be masked if B replacement node is fed starting from tuple $t0$. Doing this, the replacement operator will first output $T1$ and later on $T2$. Notice that, even if $T1$ has been received by node C, B replacement node can still be fed starting from tuple $t0$, as far as the duplicate tuple $T1$ is not being forwarded to the operator of node C.

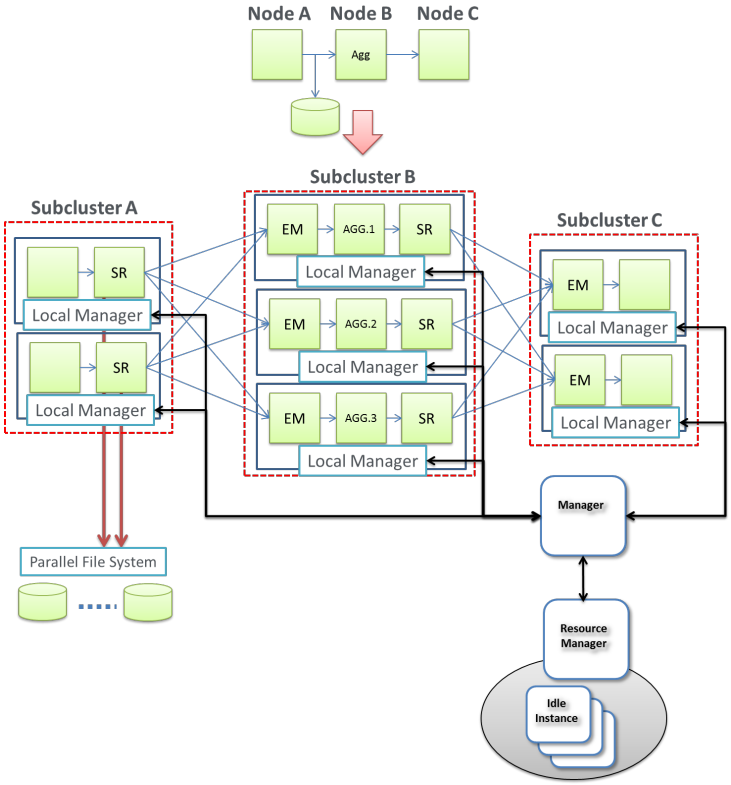

Figure 7.2: The fault tolerance architecture of CEP.

We denote as the earliest timestamp (et) of an operator the smallest timestamp from where tuples should be replayed in case of failure of operator an OP. With respect to the previous example, if $T1$ has been received by node C after node B failure, then $Agg.et$ = 180 (i.e., tuples should be replayed starting from timestamp 180 in order to provide precise recovery). On the other hand, if $T1$ has not been received by node C after node B failure, then $Agg.et$ = 0 (i.e., tuples should be replayed starting from timestamp 0 in order to provide precise recovery). It can be noticed that the earliest timestamp of an operator depends on both the tuples maintained at its state and the tuples received by its downstream peer. More precisely, the earliest timestamp represents the earliest tuple timestamp maintained by an operator if all the tuples that have been previously produced and that depend on earlier tuples have been received by the downstream instance.

7.2 Components involved in the Fault Tolerance protocol

In this section, we present an overview of the CEP components used to provide fault tolerance to query operators. Figure 7.2 resembles the sample aggregate operator presented in Figure 7.1, presenting how the operator and its upstream and downstream peers are parallelized

Following the intra-operator parallelization strategy [27], a subquery is defined for the aggregate operator while a separate subquery is defined for its preceding operator (independently of the operator type, stateless or stateful). For the ease of the explanation, assume the operator following the aggregate

is stateful, so that a dedicated subquery is defined for it. The instances of the subcluster containing the aggregate define an event merger (EM) and a semantic router (SR). Tuples are routed to the subcluster by the upstream SRs while they are collected by the downstream EMs. As discussed previously, the instances rely on their upstream peers to maintain past tuples. Particularly, the SRs persist to disk tuples while forwarding them in parallel. As shown in the figure, tuples are persisted to a parallel file system. While SRs are being employed to persist tuples, the EMs of the downstream peers are used to maintain the earliest timestamp of the operator for which we provide fault tolerance. Each CEP instance runs a Local Manager that is used to share information with the CEP Manager and to modify the running query. With respect to fault tolerance, Local Managers are used to forward the information related to the tuples being persisted and the earliest timestamp of operators to the CEP Manager.

7.3 Fault Tolerance protocol

In this section, we provide a detailed description of the fault tolerance protocol. We first focus on single instance failures and, subsequently, extend the protocol to multi-instance failures. As discussed in [27] and [29], the minimum data distribution unit used to route tuples across two subclusters is the bucket. In the following, we discuss protocols using buckets b as the minimum of tuples for which we provide fault tolerance (i.e., we discuss how to provide fault tolerance for individual buckets or groups of them). As discussed in the previous section, the earliest timestamp that specifies which tuples should be replayed in case of failure depends on the operator running at the instance we are protecting against failures. For each type of operator (i.e., stateful or stateless) and for any query that can be deployed at a CEP instance, we must define how the overall earliest timestamp is computed.

Stateless operators do not maintain any tuple; hence, we define the earliest timestamp of the operator as the timestamp of the tuples being forwarded. This means that, upon failure, we plan to resume tuples processing starting from the last tuple sent by the operator (if the latter actually reached the downstream instance, we will need to discard the duplicate tuple). With respect to stateful operators, the earliest timestamp is set to the timestamp of the earliest tuple maintained by the operator. With respect to the possible operators deployed at a given CEP instance, we discussed in [27] and [29] that according to our intra-operator parallelization strategy each subquery contains no more than one stateful operator. The earliest timestamp must be computed by the operator maintaining the earliest tuple. For this reason, if the subquery contains a stateful operator, the latter is used to compute the earliest timestamp, otherwise, the earliest timestamp is computed by the last stateless operator registered in the subquery.

In the following, we denote as I_F the instance for which fault tolerance is provided and as I_R the instance replacing the former upon failure. I_F upstream subcluster is referred to as U while downstream subcluster is referred to as D. Given a tuple T outputted by I_F , b_{et}, referred to as bucket B earliest timestamp, represents the earliest tuple timestamp of bucket b once T is produced. Each output tuple T schema is enriched with field et, set to the value of bucket earliest timestamp ($T.et = b_{et}$). Hence, if I_F fails after T has been received by D, we can recreate I_F lost state replaying its input tuples starting from timestamp $T.et$. The CEP detects that an instance I_F has failed if it stops answering to a series of consecutive heart-beat messages. Once failure has been detected, a replacement instance I_R is allocated by the Resource Manager and the query previously deployed at I_F is re-deployed at I_R. In order to know which tuples should be replayed in case of failure, b_{et} is continuously updated by the stateful operator of I_R (or the first stateless one if no stateful operator is defined) and communicated to D using output tuples.

This fault tolerance protocol ensures that, upon failure of the instance owning bucket b, b tuples are

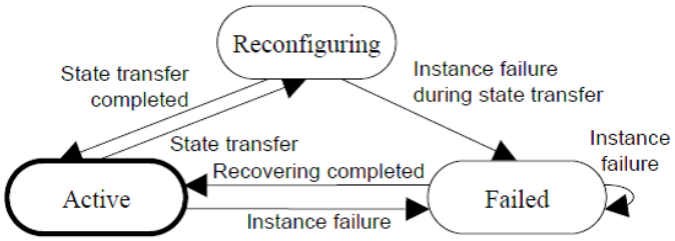

Figure 7.3: Bucket state machine.

replayed starting from a timestamp $ts <= b_{et}$ and that duplicate tuples are discarded, therefore providing precise recovery. In order to recover b in case of a failure, the protocol must (a) persist past tuples in order to replay them in case of failure, and (b) maintain the latest value of b_{et}. In our protocol, task (a) is performed by $U\ SRs$ (denoted as $U.SRs$) while task (b) is performed by $D\ EMs$ (denoted as $D.EMs$).

Figure 7.3 presents the possible states that define each bucket b. During regular processing, b state is set to Active. If b ownership is being transferred (e.g., a provisioning, decommissioning or dynamic load balancing action is triggered), its state moves to Reconfiguring. Once the reconfiguration has been completed, its state is set back to Active. Notice that reconfiguration actions are taken only for Active buckets. Failures might happen for a bucket in Active or Reconfiguring state. If failure happens while b is Active, its state moves to Failed. Bucket b remains in this state until recovery ends (i.e., it remains in Failed state also if a second failure takes place before the first one has been completed solved). b state is moved to Failed also if the failure occurs while b is in Reconfiguring state. In the following sections, we present which are the tasks performed by I_F and by its upstream downstream peers U and D depending on b state. We refer the reader to deliverable [28], presenting the State Recreation protocol, the state transfer protocol for which fault tolerance is provided in the MASSIF CEP engine. Table 7.2 summarizes the principal variables used in the following algorithms.

7.3.1 Active state

While b state is Active, $U.SRs$ are responsible for forwarding and persisting tuples being sent to I_F, the stateful operator (or the last stateless one) deployed at I_F is responsible for computing the instance earliest timestamp and use it to enriching output tuples (setting field $T.et$) while $D.EMs$ are responsible for maintaining b_{et}. With respect to the task performed by $U.SRs$, the protocol must define an efficient way to persist and forward tuples (i.e., it cannot be blocking, tuples must be forwarded and persisted in parallel). Furthermore, persisting individual tuples might result in a high overhead; hence, SRs should first buffer them and, subsequently, persist at once multiple tuples. In the proposed solution, $U.SRs$ use a buffer Buf to maintain tuples being forwarded. Each incoming tuple is added to the buffer that is persisted periodically. Similarly to time-based windows, buffers define attribute size to specify the extension of the time period they cover. Nevertheless, the periods of time covered by Buf do not overlap, they rather partition the input stream of each SR into chunks of size time units. All the SRs at U share the same $Buf.size$ and have aligned buffers (i.e., at any moment, all the buffers cover the same period of time). Given an input tuple t, and being $t.ts$ its timestamp (expressed in seconds, or other time units, from a given date), the buffer to which t belongs to will have boundaries:

Variables	
Buf	Buffer used by SRs to maintain forwarded tuples
BR	Bucket Registry
$BR[b].state$	State of bucket b
$BR[b].et$	Bucket b earliest timestamp
$BR[b].last\_ts$	Bucket b latest tuple timestamp
$BR[b].dest$	Instance to which bucket b tuples are forwarded
$BR[b].owner$	Bucket b owner
$BR[b].buf$	Buffer associated to bucket b
I_F	Failing instance
Q	Query deployed at I_F
I_R	Replacement instance
U	I_F upstream subcluster
$U.SRs$	U semantic routers
$U.SR\_PREFIX$	Name prefix shared by SRs at U
D	I_F downstream subcluster
$D.EMs$	D event mergers

Table 7.2: Variables used in algorithms

$$[\frac{t.ts}{Buf.size}, \frac{t.ts}{Buf.size} + Buf.size[$$

We refer to the left time boundary of buffer Buf as $Buf.start\_ts$. Each time the buffer is full (i.e., $t.ts - Buf.start\_ts >= Buf.size$), it is asynchronously written to disk. SRs rely on a parallel-replicated file system to persist Buf. The reason is twofold:

1. File system replication prevents information loss due to disk failures.
2. In distributed file system, tuples persisted by an SR can be accessed also by the other CEP instance running the query on other nodes.

Algorithm 1 presents SRs protocol. Each SR maintains a Bucket Registry (BR). For each bucket b, $BR[b].state$ defines b state while $BR[b].dest$ defines the instance to which b tuples are forwarded. For each incoming tuple belonging to bucket b, function buffer is invoked to add the incoming tuple to Buf and, if the buffer is full, to persist it. Subsequently, if b state is Active, t is forwarded to its destination instance. Given an incoming tuple t, function buffer checks whether Buf should be persisted and, eventually, stores tuple t. Buf is serialized if $t.ts - Buf.start\_ts > Buf.size$. The file to which the buffer is persisted is identified by the SR name persisting it and $Buf.start\_ts$. Considering how SRs persist their incoming streams, each incoming tuple is either maintained in the SR memory (if it belongs to the current buffer) or written to disk. Nevertheless, this consideration might not hold if, upon failure of

instance I_F , $U.SRs$ continue persisting the tuples they are now buffering. To avoid this, function buffer stops serializing Buf if one (or more) of the downstream instances fails. That is, tuples are still being forwarded to the active instances and buffered at Buf, but not persisted to disk. It can be noticed that, in Algorithm 1 - line 8, the condition checked by the buffer function in order to persist Buf makes sure no bucket b is in Failed state (i.e., no downstream instance has failed). The overhead introduced by the persistence of tuples is negligible. It is only caused by the time spent to create a copy of each incoming tuple and, whenever Buf is full, by the time it takes to issue the asynchronous write request. $D.EMs$ are responsible for maintaining bucket b earliest timestamp and for discarding duplicate tuples.

Algorithm 2 presents EMs protocol. For each incoming tuple $t \in b$, b earliest timestamp is updated to $t.et$. Similarly to SRs, each EM maintains a Bucket Registry BR for each bucket b. $BR[b].et$ defines b earliest timestamp, $BR[b].last\_ts$ the timestamp of the latest tuple received and $BR[b].buf$ buffer is used to temporarily store incoming tuples. If we consider how to detect duplicates, the assumption about timestamp ordered input streams and the timestamp sorting guarantees provided by the EMs, permit to spot a duplicate when its timestamp is earlier than the previous tuple or, if equal, if the tuple is a copy of another tuple sharing the same timestamp. $D.EMs$ use $BR[b].buf$ to temporarily store all the tuples sharing the same timestamp (the buffer is empty each time a new incoming tuple has a new timestamp) in order to check for duplicated tuples. Function $isDuplicate$ of Algorithm 2 presents the pseudo-code used to check for duplicated tuples.

Algorithm 1: Active State - SRs protocol.

```
1:  Upon: Arrival of tuple t ∈ b
2:  buffer(t)
3:  BR[b].dest = BR[b].owner
4:  if BR[b].state = Active then
5:      forward(t,BR[b].dest)
6:  end if
7:  function buffer(t)
8:  if ∄b : BR[b].state = Failed && t.ts >= Buf.start_ts − Buf.size then
9:      fileName = concatenate(SR.name,Buf.start_ts)
10:     async(Buf,fileName)
11:     Buf.clear()
12: end if
13: Buf.add(t)
```

We have discussed which are the protocols for $U.SRs$ and $D.EMs$ with respect to b Active state. If we analyze the interaction between these components from a subcluster point of view, we have that U serializes all the tuples forwarded to I_F , partitioning them on a per time period, per SR basis. At the same time, D maintains b_{et} on a per EM basis. If, due to a failure of I_F, the lost state must be recreated starting from timestamp ts, the tuples will be read from the files persisted by $U.SRs$. More precisely, the files to read will be the ones having name $[lb][start\_ts]$, where lb is the name of any of $U.SRs$ and:

$$start\_ts = max_i T_i : \frac{T_i}{Buf.size} <= ts$$

The tuples being forwarded by I_F and carrying information about b_{et} are routed to the different $D.EMs$ instances. At any point in time, given a bucket b, b_{et} is computed as the $min(b_{eti}); \forall EMi \in D.EMs$. That is, the earliest timestamp is the smallest one maintained by any $D.EMs$.

Algorithm 2: Active State - EMs protocol.

```
1: Upon: Arrival of tuple t from stream i
2: buffer[i].enqueue(T)
3: if ∀ i buffer[i].notEmpty() then
4:    t0 = earliestTuple(buffer)
5:    b = getBucket(t0)
6:    if ¬ isDuplicate(t0, b) then
7:       forward(t)
8:       BR[b].et = t.et
9:    end if
10: end if
11: function isDuplicated(t,b)
12: result=false
13: if t.ts > BR[b].last_ts then
14:    result = true
15: else if t.ts=BR[b].last_ts then
16:    if BR[b].buf.contains(t) then
17:       result=true
18:    else
19:       BR[b].buf.add(t)
20:    end if
21: else
22:    BR[b].last_ts = t.ts
23:    BR[b].buf.clear()
24:    BR[b].buf.add(t)
25: end if
26: returnresult
```

Bucket earliest timestamps maintenance and cleaning of stale information. Two important aspects related to the actions taken by the operators involved in the maintenance of bucket b while in state Active must be considered. If tuples are continuously persisted, they will eventually saturate the capacity of the parallel file system. Moreover, if an instance of D fails, its information associated to b_{et} will be lost. To address both problems, the CEP Manager periodically connects to D instances and retrieves the earliest timestamps of bucket b. With this information, files that only contains tuples having timestamps earlier than b_{et} can be safely discarded. Furthermore, upon failure of I_F , the earliest timestamp indicating which tuples should be replayed is known by the CEP even if one or more D instances are not reachable (e.g., due to a multiple-instance failure). It should be noticed that, even if b_{et} is not updated to its latest value, the recovery will still be precise. Algorithm 3 presents the CEP Manager protocol.

Algorithm 3: Active State - CEP Manager protocol.

1: Upon: Monitoring period expired for subcluster C
2: **for all** b **do**
3: $BR[b].et$=getBucketET(b)
4: **end for**
5: $T = min(BR[i].et/Buf.size$
6: **for all** file $[SR][ts] : SR$.hasPrefix(SRprefix) && ts ¡ T **do**
7: remove file
8: **end for**
9: function getyBucketET(b)
10: b_{et}=∞
11: **for all** em $\in$ D.EMs **do**
12: b_{et} = min(b_{et},em.BR[b].et)
13: **end for**
14: return b_{et}

7.3.2 Failed state

In this section, we present the main steps performed by the CEP to replace instance I_F in case of failure. We first discuss the overall sequence of steps and proceed then with a detailed description of each one. The failure of an instance I_F is discovered by the Manager when the former stops answering a given number of consecutive heart-beat messages. At the same time than the Manager, I_F upstream subcluster U discovers the instance has failed as soon as the TCP connection between them fails. A replacement instance I_R is taken from the pool of available instances maintained by the Resource Manager and the query previously deployed at I_F is re-deployed at I_R. While deploying the query, the lost state is recreated reprocessing past tuples persisted to the parallel file system. Once the query has been deployed, its state has been recovered and operators have been connected to their upstream and downstream peers, upstream SRs are instructed by the Manager to forward buffered tuples and resume regular processing. As soon as I_F failure is detected by $U.SRs$, the state of each bucket b owned by I_F is changed to Failed. As presented in the previous section, this implies that tuples that were previously persisted to the parallel file system by $U.SRs$ are now only maintained using main memory (as long as the failure has been recovered). SRs pseudo-code for the action performed upon failure is presented in Algorithm 4, lines 1-4.

Algorithm 4: Failed State - SRs protocol.

```
1: Upon Downstream instance I failure
2: for all b:BR[b].dest=I do
3:    BR[b].state=Failed
4: end for
5: Upon Downstream instance I recovered
6: for all b:BR[b].dest=I do
7:    BR[b].state=Active
8:    T = Buf.get(b,ts)
9:    for all t ∈ T do
10:      forward(t,BR[b].dest)
11:   end for
12:   resume regular processing
13: end for
```

Algorithm 5 presents the steps followed by the CEP Manager after discovering instance I_F has failed. We refer to the earliest timestamp from which tuples will be replayed as et. Timestamp et will be computed as the earlier among the earliest timestamps of any bucket b previously owned by I_F (5, lines 1-6). Once et has been computed, the Resource Manager is instructed to deallocate instance I_F and to allocate a replacement instance I_R. Once I_R has been allocated, the query previously deployed at I_F is re- deployed at I_R (5, lines 7-9). In order to recreate the lost state, the EM deployed at I_R is instructed to replay persisted tuples belonging to I_R buckets, starting from et. Once the lost state has been recovered and I_R has been connected to its upstream and downstream peers, $U.SRs$ are instructed to forward any buffered tuple and resume regular processing (5, lines 10-13). As shown in 4, lines 5-13, each SR changes the state of I_R buckets back to active, gets buffered tuples and, for each of them, forwards it if it belongs to I_R buckets. Once tuples have been replayed, $U.SRs$ resume regular processing.

Algorithm 5: Failed State - CEP Manager protocol.

```
1: Upon Instance I failure
2: et = ∞
3: for all b ∈ I_F do
4:    et=min(et,getBucketET(b))
5:    RB.add(b)
6: end for
7: release(b)
8: I_R = allocate()
9: deploy(Q,I_R)
10: I_R.EM.replay(ts,RB,U.SR_PREFIX)
11: for all b ∈ I; sr ∈ U.SRs do
12:    sr.recovered(b)
13: end for
```

Algorithm 6 presents the pseudocode for the EM deployed at I_R. In order to replay RB tuples, event merger at I_R first looks for all the Buf units persisted by upstream SRs and containing tuples t having timestamp $t.ts >= et$. Each of these files contains tuples persisted in timestamp order. As

multiple timestamp ordered files are read, the EM will have to merge-sort them in order to create a unique timestamp order stream of tuples. Tuples forwarded to I_F are persisted by $U.SRs$ on a per-load balancer, per-time basis. This implies that, when reading them in order to recreated I_F lost state, tuples belonging to buckets that were not owned by I_F must be discarded.

Algorithm 6: Failed State - EMs protocol.

```
1: Upon replay(ts,RB,SR_PREFIX)
2: fileNames = getFileNames(ts,SR_PREFIX)
3: F=read(fileNames)
4: mergedSort(F)
5: for all t ∈ F : t ∈ RB do
6:     forward(t)
7: end for
```

7.3.3 Failed while reconfiguring state

This section presents how the CEP fault tolerance protocol deals with failures happening during reconfiguration actions (i.e., while load is being balanced or a provisioning or decommissioning action has been triggered). In the following, we analyze separately how the fault tolerance protocol covers the failure of each instance involved in a reconfiguration action. We refer to a reconfiguration action where a bucket is being transferred from instance A to B. If, during a reconfiguration action, one of the upstream SRs fails, the failure can happen before or after all the control tuples that trigger the bucket ownership transfer have been received by A. In the former case, the reconfiguration action is postponed after recovering the failed SRs (the instances involved in the reconfiguration will not transfer any state due to the fact that they did not receive all the control tuples). In the latter case, no extra action must be taken. When recreating the failed SR at the replacement instance, it will be instructed to send incoming tuples to B (instance A will have sent the bucket state to B already, as they both received all the control tuples). In case of failure of instance A, we can identify two possible cases: the failure happens before or after the bucket state has been sent (entirely) to B. In the former case, B will not receive the state of the bucket being transferred. To solve this problem, the CEP Manager will instruct B to recreate the bucket building its state starting from the persisted tuples (fault tolerance protocol allows for the recovery of individual buckets). In the latter case, when the state has been already sent by instance A, the reconfiguration is not affected by instance A failure. State transfer is not affected if the failing instance is B. All buckets owned by B must be recovered, both if they were already owned and if they were being transferred when the instance failed.

7.3.4 Recovering state involved in previous reconfigurations

In this section, we analyze how past reconfiguration actions of the failed instance upstream subcluster triggered in the time interleaving the earliest timestamp and the failure affect recovery. That is, being t_last the last tuple processed by a failed instance I_F and being et the earliest timestamp from where to replay tuple, we analyze how reconfiguration actions involving I_F upstream subcluster taken during the period $[et; t\_last]$ affect its recovery. If I_F upstream subcluster has changed its size during period

$[et; t\_last]$ (e.g., the number of instances has decreased), then part of the tuples to replay has been persisted by an SR that does not longer exists. Nevertheless, this does not affect the protocol. As presented in Algorithm 6, the EM deployed at replacement instance I_R detects which files must be read depending on timestamp ts and SRs prefix name SR_PREFIX. Due to the fact that all the SRs of I_F upstream subcluster share the same prefix name, the files previously persisted by an SR that no longer exists are still considered in order to re-build I_F lost state.

7.3.5 Multiple instance failures

This section presents how the fault tolerance protocol deals with multiple instance failures. As we discussed, fault tolerance is provided for a given instance relying on its upstream and downstream peers. Hence, if the two instances involved in the failure are not consecutive (i.e., one subcluster is the upstream of the other), their recovery can be executed in parallel as actions triggered by the Manager will not interfere. Similarly, the recovery of two failed instances is executed in parallel if the two instances belong to the same subcluster. Opposite to these cases, if two failing instances I_{F1} ; I_{F2} belong to consecutive subclusters (I_{F1} preceding I_{F2}), their recovery must be conducted in a specific order. The deployment of I_{F1} operators and the connection to their upstream peers is executed in parallel with the deployment of I_{F2} operators and the connection to their downstream peers. Nevertheless, connections between I_{F1} and I_{F2} can be established only after the operators of each instance have been recovered. This synchronization overhead does not significantly affects the recovery time. This is due to the fact that, among all the actions taken to recover a failed instance, connection to upstream and downstream peers takes a time negligible with respect to buckets recovery actions.

8 Resilient Event Storage

This chapter presents an intrusion and fault tolerant data storage facility, called Resilient Event Storage (RES) [2] [3]. The RES is designed to ensure integrity and unforgeability of events to be stored even if some components of architecture are compromised.

According to Least Persistence principle, the only events that are permanently stored by the RES are the ones generated by a Complex Event Processing (CEP) engine at the time of the security breach and that are helpful to identify the reasons of the breach.

The RES is the core component in charge of performing correct forensic analysis when a security breach occurs. The goal of forensic analysis is to provide evidences to be used as valid proofs in a legal proceeding. The evidences collected are considered valid as proofs only if the events are not forged.

In the MASSIF project, the RES system has been integrated in two SIEM products, i.e., OSSIM and Prelude. Integration details and test results performed have been provided in deliverables [32] [33]. In particular, deliverable [33] shows that the RES works correctly even if a node of the architecture is compromised by an attacker.

8.1 Problem statement

A forensic storage in a SIEM helps retain digital evidence against malicious parties responsible for security breaches. Many free versions of SIEM solutions [4] do not implement any technique to forensically store the security events, although some of them offer this service in their commercial version. Having a forensic storage means to create an infrastructure capable of ensuring the integrity and unforgeability of stored data.

Other SIEM solutions provide a forensic storage based on a module that signs security events using classic algorithms as RSA [12]. Security events are signed and then stored in the storage system. The strength of RSA is based on the problem of factorization of large numbers which is a hard problem and no algorithm exists to solve it in real time. From dependability system point of view the signing module represents a single point of failure. In fact, if the component that signs the security events is violated for any reason, then the signing process of security events is stopped. Furthermore, the component that signs security events can be attacked from malicious users, e.g., through a DDoS attack in order to deny the service offered. A more complex attack is when the attacker steals the secret key that the component uses to sign security events. In this case the attacker can forge the signature generated while the component does not show any sign of failure.

For these reasons, the storage system must be designed to ensure unforgeabilty and integrity of data even when some components of the system are compromised. In particular, it must be resilient to attacks and work correctly even if some secret keys are stolen or compromised.

8.2 Threshold cryptography

Threshold cryptography is not a crypto-system itself but it is a technique that uses already existing crypto-systems with minor changes. These changes include the distribution of secret key into different shares. It provides more flexibility to the signing process due to the fact that this scheme works correctly also when some secret shares are compromised.

In 1979, Adi Shamir [95] and Blackley [17] proposed independently the first secret sharing scheme. According to this scheme the secret information can be divided in different parts in such a way that all or a certain number of these shares are necessary to construct the secret. Formally, let S be the secret to be split among n participants called shareholders. We want S divided in n shareholders in such a way that $k < n$ shareholders are necessary to construct S but no fewer than k shareholders can. In particular an adversary who gets control of at most $k-1$ shares has zero knowledge of the secret key. Such a scheme is called a (k, n) threshold scheme. The threshold number k is defined such that it is impossible to build the secret if less than $k-1$ shareholders cooperate together.

In this paper we have used the RSA threshold signature scheme proposed by Shoup [96] because while the previous work is theory oriented, this work is the first practical scheme that provides also implementation details. It is explained mathematically how to generate the secret key shares (SKS) from the secret key, the verification key shares (VKS) and complete verification key (VK). The SKSs can be used by different shareholders in order to create Partial Digital Signatures of a message. Moreover, the output of a digital signature algorithm in its threshold mode must be equivalent to the digital signature produced when this algorithm is used in a traditional way. Shoup has also explained the mathematical operations necessary to put together the signature shares so to obtain a complete signature equivalent to the standard RSA signature. The VKSs provide a way to check the correctness of each signature share whereas VK is used to check the validity of complete signature.

8.3 Cryptography techniques overview

In this Section three techniques to sign security events are analyzed in order to evaluate the capabilities of each one to ensure integrity and unforgeability of data to be stored. The techniques analyzed are: RSA classic scheme, parallel RSA scheme and Threshold Cryptography scheme. For parallel scheme we mean multiple RSA modules deployed in parallel. The techniques proposed can be used to carry out the RES system.

The first technique which uses classic RSA algorithm to sign the security events may face a single point of failure problem. In fact, an attacker could perform a DoS attack if he knows the IP address of the module based on classic RSA algorithm, so to stop it from signing the events. The attacker will be able to bring down the system. Also, an attacker could compromise the node that generates RSA signatures through a malicious software installed on this node and can forge the signatures generated. In this case, the RSA module works but in a wrong way so the integrity of data stored is not ensured. Also, these data are unacceptable for forensic purposes. The advantage to use the RSA classic module is that it is fast due to the fact that it does not involve huge usage of memory. In fact, when the module receives the input, it provides an output (signature) without maintaining any intermediate information.

A simple variant of traditional RSA can include the usage of multiple RSA modules working in parallel to sign the same message. All RSA modules use different public and secret key pairs. Each module signs the incoming security event with its own private key and sends the signature to an elector

element. In order to improve the security level the elector must receive at least k (threshold number) signatures for the same event before verifying the integrity of data.

When at least k (quorum) signatures arrive, the elector verifies them through their corresponding public keys and when the quorum of valid signatures is reached the elector stores the signatures and original data in the storage media. The parallel RSA scheme is slower than the classic RSA scheme because the elector, in this case, must maintain an internal state for each security event. The state related to each event is represented by the signatures received that are less than k. In this case the elector also cannot store the event with the selected signatures. Another performance penalty compared to the classic RSA scheme is due to the process to verify each received signature. In fact this process is repeated k times for each security event.

The advantage of multiple RSA modules deployed in parallel over single RSA module is that the former is tolerant to intrusions and faults. With the attack scenarios described above for single RSA module, this new architecture works correctly. In fact if one RSA module is corrupted by the attacker it will send a wrong signature to the elector. The elector simply verifies and discards this wrong signature and waits for next one. Instead, when module is under a DoS attack, the architecture also works correctly because other modules still send their signatures to the elector.

Finally we have analyzed the Threshold Cryptography scheme. In this scheme the secret key is divided in different shares and given to different parties or shareholders where each shareholder signs the incoming message with its secret share and sends the signature shares to another module called combiner. The combiner element combines all the signature shares and produces a complete signature. Then, the combiner verifies the complete signature and stores it in the storage media after successful verification. When compared to the classic RSA scheme, threshold cryptography has the same advantages and disadvantages of RSA parallel scheme as previously described. In particular, compared to RSA parallel mode, threshold cryptography is faster because in the former scheme the elector must verify at least k signatures of the same message and store them with the message. On the other hand, in threshold cryptography only complete signature must be verified. This is because at first the combiner combines the signature shares and then it verifies only the complete signature. These considerations show that threshold cryptography provides intrusion and fault tolerant capabilities that are required by RES system.

8.4 Architecture

The RES system provides an intrusion and fault tolerant data storage facility, designed to ensure the integrity and unforgeability of data to be stored even when some parts of the system are compromised. The RES architecture is shown in Figure 8.1. The inspiring principle is to use more than one secret key. In particular, we use a single secret key that is divided in n parts or shares as described in Section 8.2. The most important feature of the threshold cryptography is that the attacker has no knowledge of the secret key if less than k secret key shares are compromised. The component that generates the n secret key shares through the secret key is called dealer. The dealer is not shown in the RES architecture because it is used only in initialization phase to generate and to distribute the secret key shares. The process performed by the dealer to generate and distribute secret key shares is shown in Figure 8.2. In particular, when the dealer is activated, it generate n secret key shares from secret key and distributes each secret key share to each node of RES system.

At steady-state, the RES receives as input the security events and information that have been collected from different sources. We assume that the event source is trusted and reliable. The input message containing information about the security incident is similarly sent to all nodes and the combiner com-

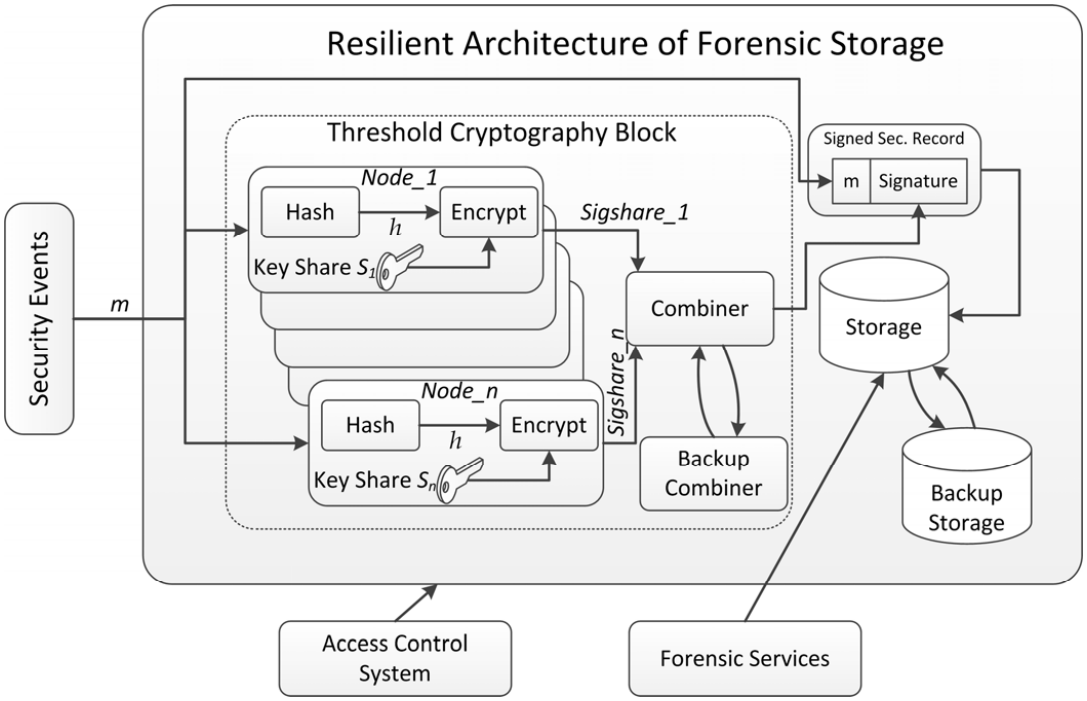

Figure 8.1: Resilient event storage architecture.

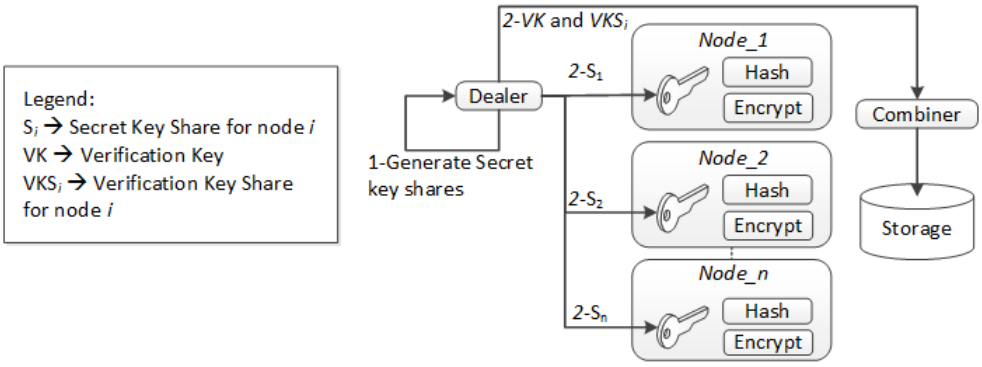

Figure 8.2: The generation and distribution of all secret key shares by the Dealer.

ponent. Each node or shareholder that has received the event computes a hash function of the message. This function returns a digest for this message, represented by h in Figure 8.1.

The next step performed by each node is to encrypt the digest with the secret key share previously sent by Dealer in order to produce a signature share (or partial signature) and send it to the combiner. The combiner is responsible for assembling all partial signatures of the same message received from the participants to the threshold crypto-system process. After verification, the complete signature generated by the combiner is attached to the original message, thus composing a signed security record.

In Figure 8.1, Forensic Services provide a forensic analysis on the data stored in the RES in order to discover information about the attacker/s. The Access Control System performs authentication and authorization tasks. For example, only the system administrator can change the threshold value k or add a new node to the architecture.

In order to enhance the fault tolerance of the RES, replication and diversity are used. In fact, the combiner and the storage are implemented through a set of software replicas, which are deployed on independent servers.

8.5 Implementation

A distributed approach has been used in the design and implementation of the RES. The basic idea is to combine the advantages of the threshold cryptography with the advantages of distributed architectures in order to obtain a system that guarantees security of data and tolerance to both intrusions and faults. In fact, the threshold cryptography guarantees the integrity of data if less than k out of n shareholders are corrupted or down for any reason. Also, the distributed system is easily scalable allowing enhancement in the performance. The components of the RES architecture are exported as a service.

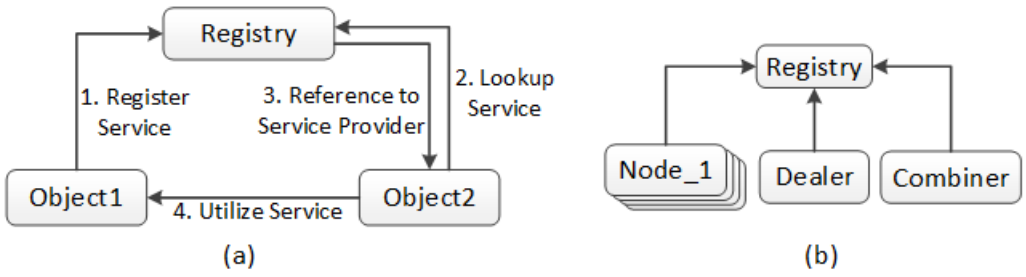

Figure 8.3: (a) An example of binding and lookup of a service; (b) RES services.

In Figure 8.3(a) the mechanism of binding is shown between a component that offers a service and a component that looks for a service.

Object1, which offers a service, registers its service in the registry. Object2 does not know anything about Object1, but it knows only the name of the service that it needs. Object2 searches for the service into the registry by using the name of the service as a search key. If the specified service exists, registry sends a reference to Object2 so that it may contact the service provider, i.e., Object1. The registry is the only component that maintains a list of all services. Therefore, it is a single point of failure of the architecture. For this reason, the registry must be designed to be tolerant to faults and intrusions.

In Figure 8.3(b) the components that offer the services needed to build the RES system are shown. In particular, the component $Node\_i$ provides a service to calculate the hash value of a message and to

encrypt it in order to generate a signature share. The service offered by dealer component is generation and distribution of secret key shares to all nodes and combiner component. Combiner component offers a service that generates a complete signature using the signature shares provided by different nodes and it verifies the complete signature.

In the following subsection more details will be provided about each component identified.

The technology used to implement the distributed architecture is Java-Remote Method Invocation (Java-RMI [84]). The Java-RMI system allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. RMI provides for remote communication between programs written in the Java programming language. RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Distributed object applications need to do the following:

- Locate remote objects. Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- Communicate with remote objects. Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

In order to improve the level of security when the different services communicate with each other, only communication through SSL [47] is allowed. The chosen authentication mode is bilateral. Certificate management in Java is made possible through the use of key-store and trust-store. All certificates have been pre-generated and recorded in the key-store and trust-store. The components that participate in the threshold signature scheme are described below.

8.5.1 System initialization phase

The sequence diagram shown in Figure 8.4 describes the interactions between RES components involved during initialization phase. The services offered by components are registered in the following order:

- The combiner component registers the service offered, i.e., combining and verification service. The combining service is used to put together the signature shares provided by nodes in order to generate a complete signature. The verification is used to check the correctness of complete signature. After registering the service offered, the combiner establishes a connection with storage system, e.g., a MySQL database.
- Each node registers the service offered, i.e., encryption service. Specifically the service offered by each node is to calculate the digest for each message and encrypt it with its own secret key

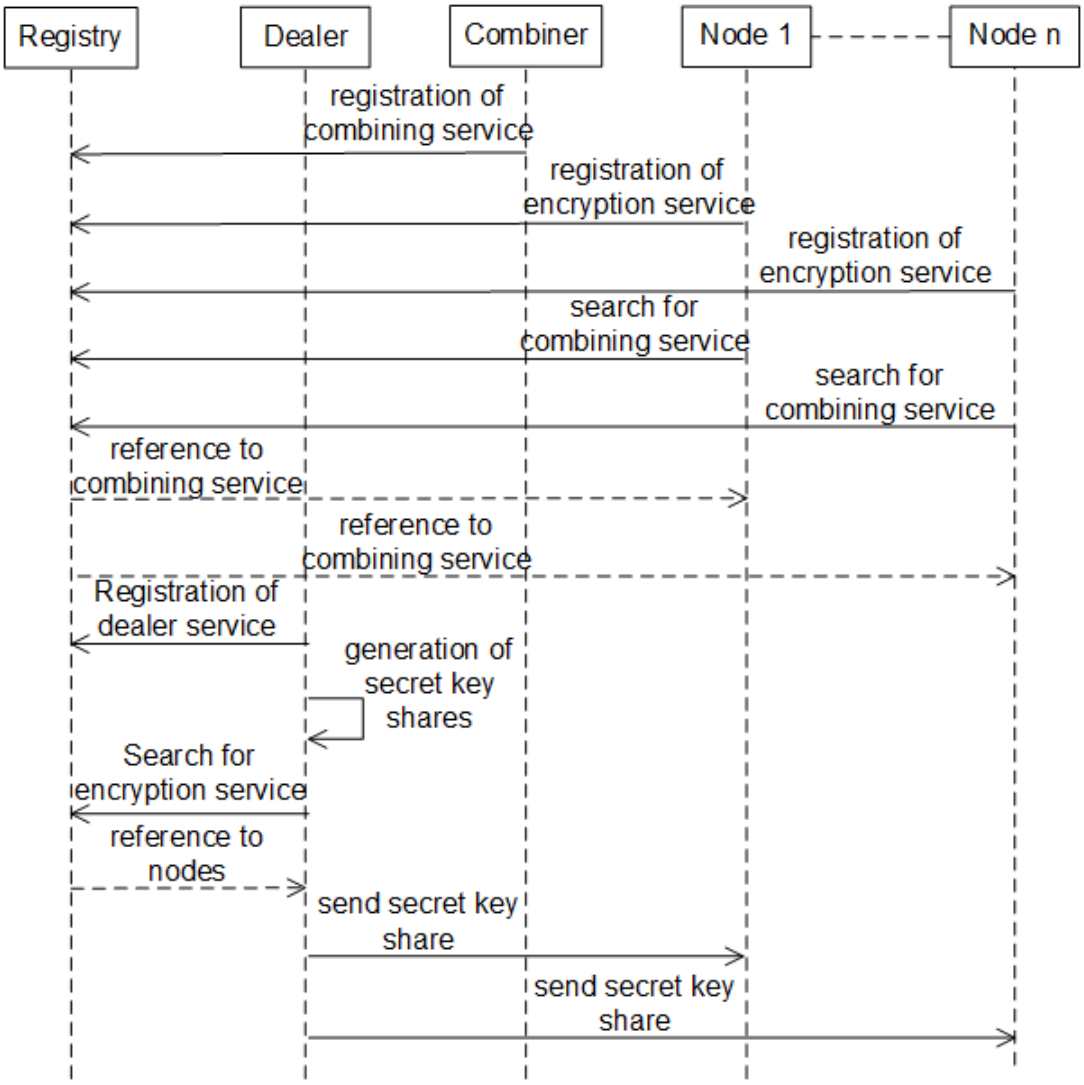

Figure 8.4: RES - Initialization phase

share. In this step the node has not received the secret key share yet. After each node registered successfully the service offered it sends a request to the registry component to know who offers a combining service. This is because at steady state the signatures generated by each node are sent to the combiner component. The registry sends each node the information to contact the component that offers a combining service, i.e., the combiner.

- The last component that performs the registration of the service is the dealer. After the registration is completed successfully the dealer generates secret key shares. Then the dealer sends a request to the registry to know the identifiers of nodes registered and how to contact them. The registry sends the list of nodes registered previously to the dealer. Finally, the dealer sends each secret key share to each node involved in the system.

The initialization phase is finished when the previous actions are completed successfully.

8.5.2 Steady-state behavior

The sequence diagram shown in Figure 8.5 describes the interactions between RES components at steady state. In particular, the sequence diagram in Figure 8.5 shows the RES behaviour both when there are not compromised nodes and when one of them is compromised. A node is considered as compromised when it sends wrong signature shares related to events processed.

- Security Event Source generates a new event both when a security breach or an anomaly is detected. The event generated is sent by nodes and combiner component.
- Each node generates a new thread in order to process the new event. Multi-thread approach allows each node to receive and process security event at once. The node calculates the digest for new event using an hash function. Then it generates a signature share by encrypting the digest through its own secret key share. Finally, the event is sent to the combiner component.
- Combiner component stores the security event sent by Security Event Source in the data structure (details on the data structured are provided in the following). Then the combiner stores the signature shares sent by each node in the data structure. When at least k signature shares are available for a specific event, the combiner puts together the signature shares and generates a complete signature. Finally the combiner verifies the correctness of complete signature.

When a node is compromised by and attacker, it starts to send forged signature shares in order to invalidate the evidences gathered. In Figure 8.5 node_1 sends a wrong signature share to the combiner after being hacked. After the combiner has received at least k signature shares for the same event it generates the complete signature and tries to verify it. Of course the verification process fails because of the wrong signature share. The combiner uses other signature shares in order to re-generate a complete signature. When the verification process is performed successfully complete signature is stored together with the raw event in the storage system.

8.5.3 Dealer

Dealer is responsible for providing a service for generation of secret key shares, distribution of these key shares to each participant or node involved in the architecture and distribution of verification key shares

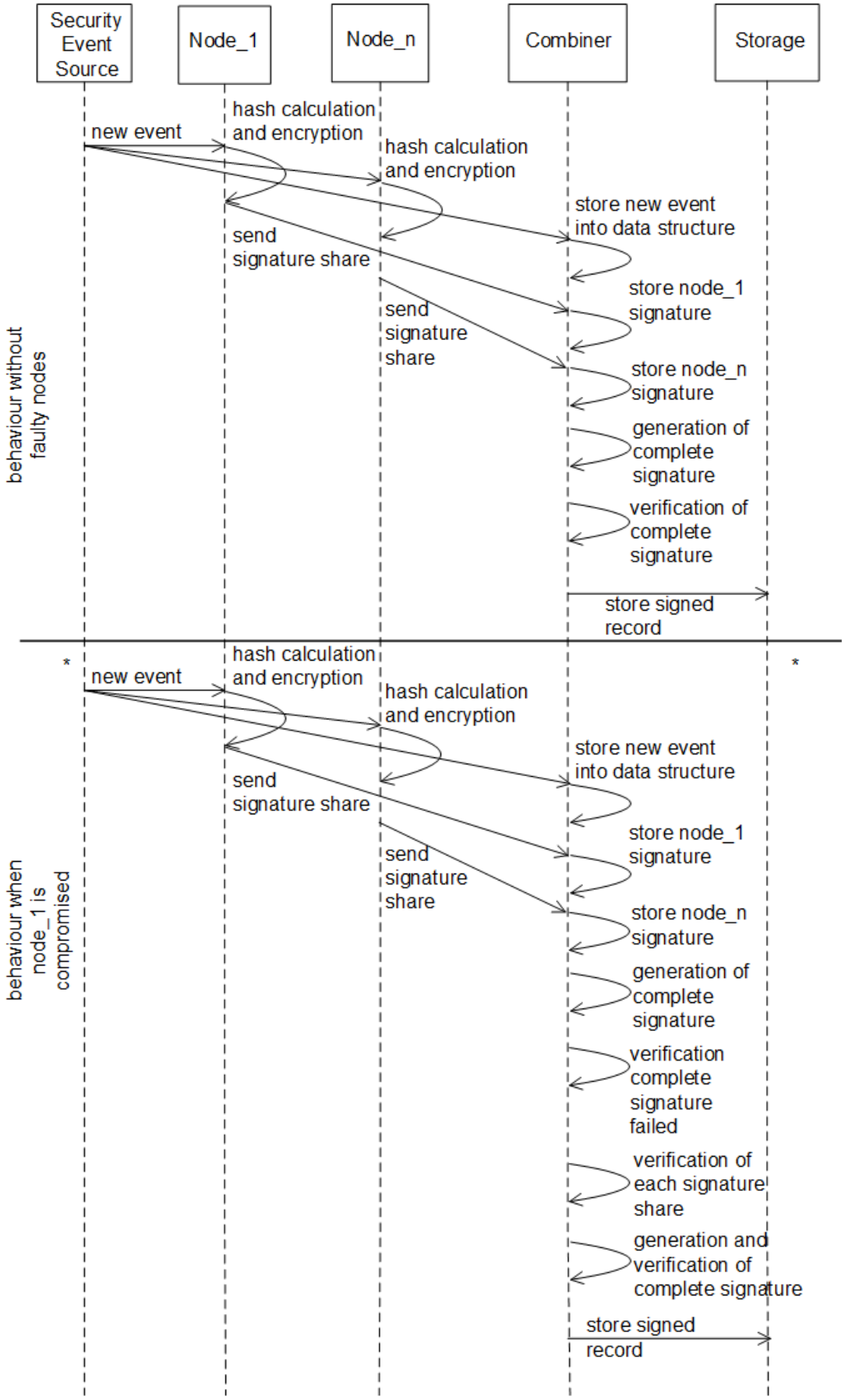

Figure 8.5: RES - Steady-State Behaviour

to the combiner component. When the Dealer is activated, the following method is called in order to generate all keys:

- *public void generateKeys (int keySize, int threshold, int groupPlayers) throws RemoteException;*

The threshold and groupPlayers are the parameters of the Shoup [96] scheme, while keySize represents the size in bits of each key share that will be generated. Before the method ends, a search is made in the Registry by the Dealer to find the Combiner component and the nodes responsible for partial signatures generation. The Registry informs the Dealer which nodes are available and also provides the way to contact them (the same applies to the Combiner). The Dealer sends each secret key share to each node. Also, it sends all the verification key shares and the verification key to the combiner component as shown in Figure 8.2. When each node has received the share of secret key, it sends an acknowledgment to the Dealer. If the Dealer receives acknowledgments from all nodes and the Combiner, it deletes all generated key shares. Otherwise a new log is generated with an exception. The erasure of the secret key shares is very important so that if the Dealer is compromised in the future, the adversary will not be able to recover the secret key shares from its memory. The Dealer component works only during the initialization phase.

8.5.4 Node

Each node performs the following tasks: when a new message arrives, it calculates the hash value of the new message and then encrypts this hash value with the secret key share. The services provided by each node are accessible through two methods:

- *public int setKey (BigInteger secretKeyShare) throws RemoteException;*
- *public void setMessage (String message, int id) throws RemoteException;*

The setKey method is used in the initialization phase. In fact when the Dealer has generated the secret key shares it distributes each secret key share to each node invoking this method. In the initialization phase, each node searches in the Registry for the service provided by Combiner component. The setMessage method is used to send a new message to the node. When a new message arrives, a new thread is created. The thread uses the hash function SHA-256 to calculate the digest value. Then, the thread uses the secret key share to generate the signature share. Finally, the thread sends the signature share and the id of the message to the combiner component. The id value is required by the combiner component in order to associate received signature shares to the correct message.

8.5.5 Combiner

In the initialization phase, the combiner performs the following actions:

- creates the link with a database where the signed events are stored
- receives all verification key shares to check the integrity of signature shares of each shareholder. It also receives the full verification key that can be used to verify the validity of the complete signature once the combining process has been completed

- creates a hash table to store temporarily all messages and their signature shares received from multiple nodes
- creates two kinds of threads, i.e., producer and consumer, which are used during subsequent phase

In the steady-state, the producer thread performs two tasks: it stores the new incoming events in the hash table according to their id and stores the signature shares sent by nodes corresponding to each event. The consumer thread works asynchronously. In fact, once it has been created, it periodically reads the whole hash table to check if it is possible to generate new complete signature. Since the time to verify all signature shares is higher than the time to verify complete signature, we always use the optimistic approach. In fact, when the combiner receives at least k (threshold) signature shares, it generates the complete signature as shown in Figure 8.6.

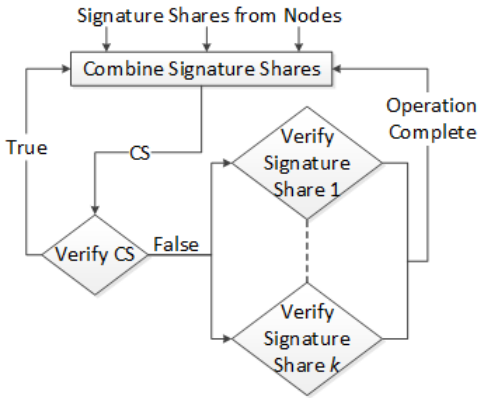

Figure 8.6: Behaviour of combiner in presence and absence of wrong signature shares

Then, it verifies the integrity of this complete signature with the verification key. If the complete signature (CS) is valid, it is stored in the database together with the original event. If the complete signature is not valid, the combiner explores in depth each signature share and checks their integrity with corresponding verification key share. If the signature share is valid a flag is set to true, otherwise the flag is set to false.

In this way, the next time when a new signature share arrives from the same event source, the combiner takes the first k signature shares excluding the signature share whose flag is set to false and generates a complete signature. If the complete signature is valid, the combiner stores in the database the event with its valid signature along with the faulty node, otherwise the process is repeated. When a faulty node is identified, the system administrator will be informed about that in order to take necessary security measures to bring it in working condition. The services provided by the Combiner are accessible through three methods:

- *public void setParameters (BigInteger verificationKeyShares[ ],BigInteger verificationKeyComplete) throws RemoteException;*
- *public void setMessage (String message, int id) throws RemoteException;*
- *public void setSigShare (BigInteger sigShare, int id, int pos) throws RemoteException;*

The setParameters method is used in the initialization phase to store all verification key shares and full verification key. The setMessage method starts the producer thread that stores the message in the hash table using id value as key. The setSigShare method starts the producer thread that adds the new signature share ”sigShare” in the hash table from node ”pos” to the message with id value equal to id.

Optimization of the data structure: The methods shown in the code above called setMessage and setSigShare work in parallel by writing in the same hash table. Moreover the consumer thread also writes a flag in the hash table if a valid signature share or a full signature is found. This problem is known as the producer-consumer problem. There are many solutions to this problem and one of them is to perform one write operation at time. We have found a new way to overcome this limitation partially. In particular, better organization of data in the data structure allows some write operations to be performed in parallel. The data structure that we have chosen is shown in Figure 8.7. Using this data

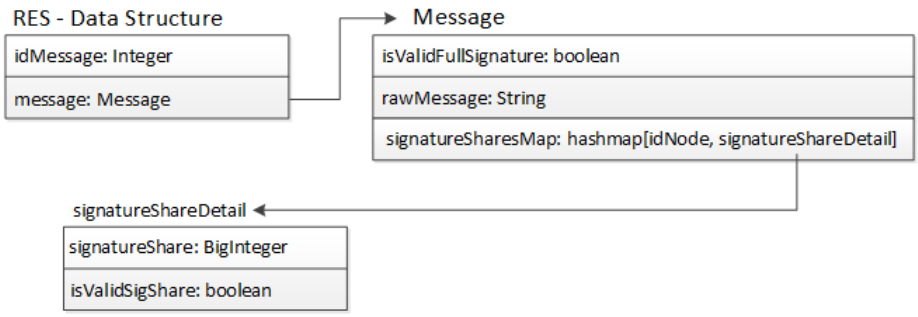

Figure 8.7: data structure used to handle all signature shares in the combiners

structure, when a new signature share is sent from a node to the combiner, the producer thread adds it to the field signatureSharesMap of a message with identifier idMessage. Since the key of the hash table signatureSharesMap is the identifier of the node that has generated the signature share, next time a write operation has to be performed the producer thread does not change this record anymore. This is because in this case each node generates only one signature share for a message. When the consumer thread checks if a signature share is valid, it can write the flag isValidSigShare directly without generating race conditions with the producer thread. This optimization allows improving the performance of the global system.

8.6 Integration with OSSIM and Prelude SIEMs

RES system was integrated in two SIEM products provided by two MASSIF partners, i.e., AlienVault and Prelude. The integration with these SIEMs [32] [33] allowed to show that RES works correctly even if some nodes are compromised. The architecture implemented to integrate RES with SIEM products is shown in Figure 8.8.

In particular the integration is performed without any modifications to SIEM source code. This was obtained analyzing the SIEM database structure. The purpose is to understand which tables store security events generated when a correlation directive is activated. The SIEM sensors can generate several events not related to real security breaches. The correlation directives are used to recognize real

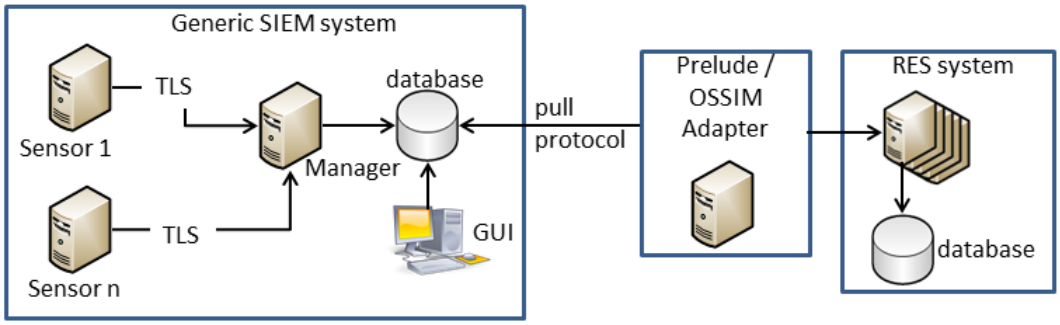

Figure 8.8: Integration Architecture between SIEM and RES systems

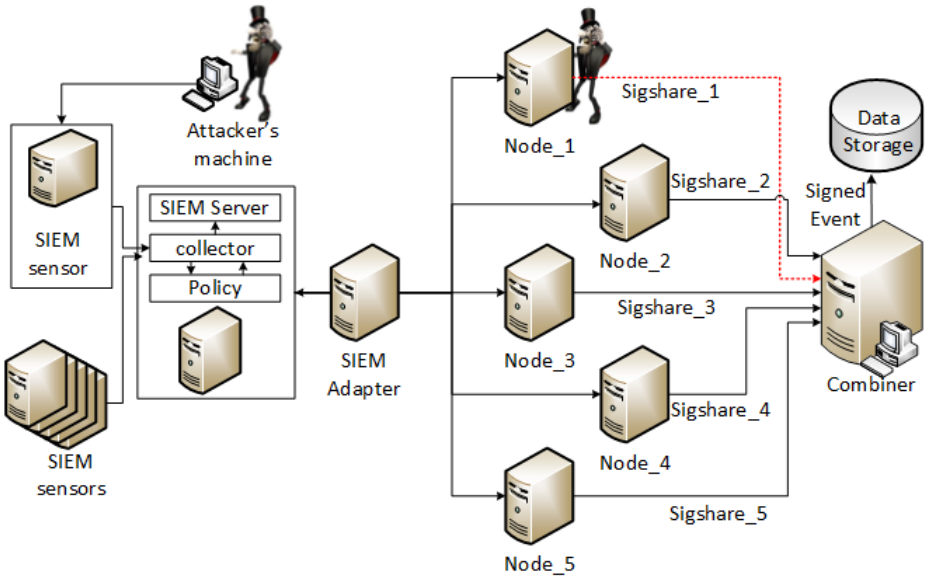

Figure 8.9: Simplified deployment of a generic SIEM with RES system

security breaches. Then, a new alert is generated when the correlation directive is activated; this alert and the chain of related events are stored in secure way by RES.

An additional component called OSSIM/Prelude adapter was developed to retrieve the alert and the chain of related events from SIEM database when a correlation directive is activated. Finally the adapter sends security events retrieved from SIEM database and sends them to RES system that processes them as described above.

A testbed was created in order to perform some tests using SIEM and RES systems. RES configuration used for test purposes includes $n = 5$ nodes and a threshold $k = 3$. With this setup it is possible to tolerate until $k - 1 = 2$ faulty nodes. The simplified architecture is shown in Figure 8.9 where an attack model is also shown. In order to simulate the faulty node, a RES node was hacked by attacker, specifically node_1 with identifier $id = 1$. Node_1 processes security events in the same way as other nodes but it forges the signature share generated modifying some bits. Finally, the same attacker performs a bruteforce attack in order to discover the administrator password of one control machine. In the attacked machine a SIEM sensor is running that gathers security events and sends them to the SIEM server. In

corruptednode	signature	correlationIde	correlationDescriptio	alertident	analyzerid	alertIdentifier	rawLog	alertDescription
1 -	10552659	220	Multiple failed logins	add04fdc-	409542446	219	Feb 26 21:47:05 localhost su: par	Credentials Change
1 -	20829058	220	Multiple failed logins	aa47fb80-i	409542446	217	Feb 26 21:46:59 localhost su: par	Credentials Change
1 -	76774716	220	Multiple failed logins	abd8ce7a-	409542446	218	Feb 26 21:47:01 localhost su: par	Credentials Change
1 -	10409347	220	Multiple failed logins	a6b2919c-	409542446	215	Feb 26 21:46:53 localhost su: par	Credentials Change
1 -	10854610	220	Multiple failed logins	a89814d2-	409542446	216	Feb 26 21:46:56 localhost su: par	Credentials Change

Figure 8.10: Logs stored in RES when Node 1 is compromised

order to recognize this type of attack a correlation directive was written. When multiple failures during change of credentials are performed the correlation directive is activated and an alert is generated. The alert generated and the chain of related events are sent to nodes and combiner component. Node_1 generates a forged signature share. Then the combiner generates and verifies complete signature. The verification process fails because of the wrong signature share. The node that sent the forged signature is identified. In Figure 8.10 the records generated by RES are shown. In particular the column ”corruptednode” contains the identifier of the node that sent a wrong signature.

9 Conclusions

This report consolidates the investigation performed within the MASSIF project towards the development of a more resilient SIEM system. Our solution addresses various concerns regarding failures on both the network and nodes, encompassing problems of accidental nature and malicious attacks. The first part of the report presents the main architectural options that contribute positively to a more dependable and secure operation of the SIEM as a whole. In the second part, we describe specific solutions for improving the resilience of some of the identified components:

Authenticated Component Event Reporting discusses mechanisms that can be employed to give evidence that produced event data has not been tampered with, and therefore,

Robust Event Reporting covers two areas: it discusses mechanisms that can be employed to give evidence that produced event data has not been tampered with, and therefore, that can effectively be used to make decisions with regard to security problems that are observed in the monitored systems; and, it describes method to hardened correlation rules, in order to address various faults that might affect (or corrupt) event collection;

Resilient Event Bus (REB) presents a solution for data dissemination from the edge-MIS towards the core-MIS. This solution achieves high levels of resilience by resorting to a number of mechanisms, such multipath data transmission, multihoming, and coding algorithms.

Node defense mechanisms explains a set of techniques that can be applied in an incremental way to make nodes increasingly more resilient to different forms of faults, of either accidental nature or malicious attacks. The design of a highly resilient core-MIS that takes advantage of the proposed techniques is also described.

Complex Event Processing describes a way of making the correlation of events in a more dependable way, for instance when crashes occur on a subset of the nodes responsible for performing the rule operations.

Resilient Event Storage (RES) presents a solution for the secure archival of event data. RES gives support for the forensic analysis of an incident based on the stored data, and enforces security policies that ensure that events are only made available to authorized entities according to the regulations.

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 59–74. ACM, 2005.

[2] M. Afzaal, C. Di Sarno, L. Coppolino, S. D'Antonio, and L. Romano. A resilient architecture for forensic storage of events in critical infrastructures. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 48–55, 2012.

[3] M. Afzaal, C. Di Sarno, S. Dantonio, and L. Romano. An intrusion and fault tolerant forensic storage for a siem system. In *Signal Image Technology and Internet Based Systems (SITIS), 2012 Eighth International Conference on*, pages 579–586, 2012.

[4] AlienVault. *OSSIM, the Open Source SIEM. http://www.alienvault.com/pricing/compare-ossim-to-alienvault-usm*. AlienVault, April 2013.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 131–145, 2001.

[7] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., 2001.

[8] J. Antunes and N. Neves. Diveinto: Supporting diversity in intrusion-tolerant systems. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems*, October 2011.

[9] J. Antunes and N. Neves. Using behavioral profiles to detect software flaws in network servers. In *Proceedings of the 22nd Annual International Symposium on Software Reliability Engineering*, pages 1–10, November 2011.

[10] J. Antunes and N. Neves. Recycling test cases to detect security vulnerabilities. In *Proceedings of the 23nd Annual International Symposium on Software Reliability Engineering*, November 2012.

[11] J. Antunes, N. Neves, and P. Verissimo. Reverse engineering of protocols from network traces. In *Proceedings of the 18th Working Conference on Reverse Engineering*, October 2011.

[12] Assuria. *A CESG CCTM Accredited Forensic SIEM/Log Management solution*. Assuria Log Manager (ALM), Assuria Ltd, April 2011.

[13] R. Baldoni, J.-M. Hélary, M. Raynal, and L. Tangui. Consensus in Byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, April 2003.

[14] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[15] A. Bessani, E. Alchieri, M. Correia, and J. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems – EuroSys'08*, pages 163–176. ACM, 2008.

[16] A. Bessani, P. Sousa, M. Correia, N. Neves, and P. Verissimo. The CRUTIAL way of critical infrastructure protection. *IEEE Security and Privacy*, 6(6):44–51, November/December 2008.

[17] G. Blakley. Safeguarding cryptographic keys. In *Proceedings of the International Workshop on Managing Requirements Knowledge*, 1979.

[18] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium*, pages 1–14, 2003.

[19] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[20] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, November 2002.

[21] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[22] J. Chauhan and R. Roy. Is Firewall and Antivirus Hacker's Best Friend? Technical report, iViZ Techno Solutions, Jan 2011.

[23] Cisco. Multiple vulnerabilities in firewall services module. http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20070214-fwsm, February 2007.

[24] Cisco. Multiple vulnerabilities in Cisco firewall services module. http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20121010-fwsm, October 2012.

[25] MASSIF Consortium. *Deliverable D2.1.1 - Scenario requirements*. Project MASSIF EC FP7-257475, April 2011.

[26] MASSIF Consortium. *Architecture Document*. Project MASSIF EC FP7-257475, April 2012.

[27] MASSIF Consortium. *Deliverable D3.1.3 - Event processing engine*. Project MASSIF EC FP7-257475, 2012.

[28] MASSIF Consortium. *Deliverable D3.1.4 - Design of elastic computing component*. Project MASSIF EC FP7-257475, 2012.

[29] MASSIF Consortium. *Deliverable D3.1.5 - Distributed event processing operators*. Project MASSIF EC FP7-257475, 2012.

[30] MASSIF Consortium. *Deliverable D3.1.6 - Elastic event processing engine*. Project MASSIF EC FP7-257475, 2012.

[31] MASSIF Consortium. *Deliverable D2.3.3 - Acquisition and evaluation of the results*. Project MASSIF EC FP7-257475, Set 2013.

[32] MASSIF Consortium. *Deliverable D5.4.2 - OSSIM Integration*. Project MASSIF EC FP7-257475, June 2013.

[33] MASSIF Consortium. *Deliverable D5.4.3 - Prelude Integration*. Project MASSIF EC FP7-257475, April 2013.

[34] L. Coppolino, M. Jäger, N. Kuntze, and R. Rieke. A Trusted Information Agent for Security Information and Event Management. In *Proceedings of the International Conference on Systems*. XPS, 2012.

[35] M. Correia, N. Neves, Lau Lung, and P. Verissimo. Low complexity byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249, 2005.

[36] M. Correia, N. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, October 2004.

[37] M. Correia, N. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, January 2006.

[38] M. Correia, N. Neves, and P. Verissimo. BFT-TO: Intrusion tolerance with less replicas. *The Computer Journal*, 56(6):693–715, June 2013.

[39] M. Correia, G. Veronese, N. Neves, and P. Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *International Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[40] A. Daidone, F. Di Giandomenico, A. Bondavalli, and S. Chiaradonna. Hidden Markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 245–256, October 2006.

[41] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[42] T. Dierks and C. Allen. The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments, January 1999.

[43] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[44] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings of the 3rd European Dependable Computing Conference*, September 1999.

[45] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322, 1988.

[46] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[47] A. Freier, P. Karlton, and P. Kocher. The Secure Sockets Layer (SSL) Protocol Version 3.0. IETF RFC 6101, 2011.

[48] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 383–394, June 2011.

[49] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. Analysis of OS diversity for intrusion tolerance. *Software - Practice and Experience*, accepted for publication.

[50] M. Garcia, N. Neves, and A. Bessani. Diversys: Diverse rejuvenation system. In *Proceedings of the INFORUM - Simpósio de Informática*, September 2012.

[51] S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security (Third Edition)*. O'Reilly Media, 2003.

[52] P. Gladyshev and A. Patel. Formalising event time bounding in digital investigations. *International Journal of Digital Evidence*, 2005.

[53] D. Grawrock. *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition, 2009.

[54] Trusted Computing Group. TCG Trusted Network Connect – TNC IF-MAP Binding for SOAP Version 2.0. www.trustedcomputing.org, 2010.

[55] Trusted Computing Group TPM Working Group. TCG Specification Architecture Overview. http://www.trustedcomputinggroup.org/resources/, 2007.

[56] K. Gummadi, H. Madhyastha, S. Gribble, K. Levy, and D. Wetherall. Improving the Reliability of Internet Paths with One-hop Source Routing. In *In Proceedings of the Symposium on Operating Systems Design & Implementation*, 2004.

[57] V. Hadzilacos and S. Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR 94-1425, Department of Computer Science, Cornell University, New York - USA, May 1994.

[58] D. Harkins and D. Carrel. The Internet Key Exchange, 1998.

[59] F. Izquierdo, M. Ciurana, F. Barcelo, J. Paradells, and E. Zola. Performance evaluation of a toa-based trilateration method to locate terminals in wlan. In *1st International Symposium on Wireless Pervasive Computing*, 2006.

[60] J. Haddix et al. HP 2012 cyber risk report. Hewlett-Packard Development Company, February 2013.

[61] S. Kamara, S. Fahmy, E. Schultz, F. Kerschbaum, and M. Frantzen. Analysis of vulnerabilities in Internet firewalls. *Computers & Security*, 22(3):214–232, 2003.

[62] R. Kissel. Glossary of key information security terms. NIST Interagency Reports NIST IR 7298 Revision 1, National Institute of Standards and Technology, February 2011.

[63] N. Kuntze, D. Mähler, and A. U. Schmidt. Employing trusted computing for the forward pricing of pseudonyms in reputation systems. In *Proceedings of the 2nd International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution, Volume for Workshops, Industrial, and Application Sessions*, 2006.

[64] N. Kuntze and C. Rudolph. Secure digital chains of evidence. In *Proceedings of the Sixth International Workshop on Systematic Approaches to Digital Forensic Engeneering*, 2011.

[65] T. Kunz, S. Okunick, and U. Pordesch. Data Structure for Security Suitabilities of Cryptographic Algorithms (DSSC)-Long-term Archive And Notary Services (LTANS). Technical report, IETF Internet-Draft, 2008.

[66] J. Lacan, V. Roca, J. Peltotalo, and S. Peltotalo. Reed-Solomon Forward Error Correction (FEC) Schemes. IETF RFC 5510, April 2009.

[67] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, 1982.

[68] M. LeMay and C. Gunter. Cumulative attestation kernels for embedded systems. *Computer Security–ESORICS 2009*, pages 655–670, 2009.

[69] J. Liu, F.R. Yu, Lung C.-H., and H. Tang. Optimal combined intrusion detection and biometric-based continuous authentication in high security mobile ad hoc networks. *IEEE Transactions on Wireless Communications*, 8(2), 2009.

[70] M. Luby. Lt codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 271–280, 2002.

[71] D. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[72] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[73] M. Marsh and F. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January 2004.

[74] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. IETF RFC 2018, October 1996.

[75] S. Mcclure, J. Scambray, and G. Kurtz. *Hacking Exposed 7: Network Security Secrets and Solutions (Seventh Edition)*. McGraw-Hill Osborne, 2012.

[76] I. Medeiros, N. Neves, and M. Correia. Securing energy metering software with automatic source code correction. In *Proceedings of the IEEE International Conference on Industrial Informatics*, July 2013.

[77] A. Mishra, B. B. Gupta, and R.C. Joshi. A comparative study of distributed denial of service attacks, intrusion tolerance and mitigation techniques. In *Proceedings of the Intelligence and Security Informatics Conference*, pages 286–289, Sep 2011.

[78] C. Mitchell. *Trusted Computing*. Iet, 2005.

[79] H. Moniz, N. Neves, and M. Correia. Byzantine fault-tolerant consensus in wireless ad hoc networks. *IEEE Transactions on Mobile Computing*, accepted for publication.

[80] H. Moniz, N. Neves, M. Correia, and P. Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, jun 2006.

[81] B. Mukherjee, L. Heberlein, and K. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, 1994.

[82] N. Neves, M. Correia, and P. Verissimo. Solving vector consensus with a wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

[83] R. Oppliger. Security at the Internet layer. *IEEE Computer*, 31(9):43–47, September 1998.

[84] Oracle. Java Remote Method Invocation (Java-RMI). http://docs.oracle.com/javase/tutorial/rmi/, July 2013.

[85] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. IETF RFC 6298, June 2011.

[86] M.M. Pollitt. Report on digital evidence. In *13th INTERPOL Forensic Science Symposium*. Citeseer, 2001.

[87] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[88] H. Reiser and R. Kapitza. Fault and intrusion tolerance on the basis of virtual machines. In *Tagungsband des 1. Fachgesprach Virtualisierung*, Feb 2008.

[89] M. Reith, C. Carr, and G. Gunsch. An examination of digital forensic models. *International Journal of Digital Evidence*, 1(3):1–12, 2002.

[90] J. Richter, N. Kuntze, and C. Rudolph. Securing digital evidence. In *Proceedings of the Fifth International Workshop on Systematic Approaches to Digital Forensic Engeneering*, pages 119–130, 2010.

[91] J. Richter, N. Kuntze, and C. Rudolph. Security Digital Evidence. In *2010 Fifth International Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 119–130. IEEE, 2010.

[92] T. Roeder and F. Schneider. Proactive obfuscation. *ACM Transactions on Computer Systems*, 28(2):4:1–4:54, 2010.

[93] F. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[94] SELinux. Security Enhanced Linux. http://selinuxproject.org/, 2013.

[95] A. Shamir. How to share a secret. *Communications of the ACM*, 22:612–613, 1979.

[96] V. Shoup. Practical Threshold Signatures. In *Proceedings of the International Conference on Theory and Application of Cryptographic Techniques*, pages 207–220, 2000.

[97] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the European Dependable Computing Conference*, pages 37–48, May 2012.

[98] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Veríssimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel Distributed Systems*, 21(4):452–465, April 2010.

[99] P. Sousa, N. Neves, and P. Verissimo. Hidden problems of asynchronous proactive recovery. In *Proceedings of the Workshop on Hot Topics in System Dependability*, June 2007.

[100] G. Starnberger, L. Froihofer, and K.L. Goeschka. Using smart cards for tamper-proof timestamps on untrusted clients. In *International Conference on Availability, Reliability and Security, ARES*, pages 96–103. IEEE Computer Society, 2010.

[101] F. Stumpf, A. Fuchs, S. Katzenbeisser, and C. Eckert. Improving the scalability of platform attestation. In *Proceedings of the Third ACM Workshop on Scalable Trusted Computing*, pages 1–10, October 31 2008.

[102] S. Surisetty and S. Kumar. Is McAfee securitycenter/firewall software providing complete security for your computer? In *Proceedings of the International Conference on Digital Society*, pages 178–181, Feb 2010.

[103] B. Vavala and N. Neves. Robust and speculative Byzantine randomized consensus with constant time complexity in normal conditions. In *Proceedings of the 31th IEEE Symposium on Reliable Distributed Systems*, October 2012.

[104] P. Verissimo, N. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the IEEE Third Survivability Workshop*, pages 157–161, October 2000.

[105] P. Veríssimo, N. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.

[106] P. Veríssimo, N. Neves, M. Correia, and P. Sousa. Intrusion-resilient middleware design and validation. In *Information Assurance, Security and Privacy Services*, volume 4 of *Handbooks in Information Systems*, pages 615–678. 2009.

[107] Vmware. http://www.vmware.com/, 2013.

[108] M.G. Wing, A. Eklund, and L.D. Kellogg. ”consumer-grade global positioning system (gps) accuracy and reliability”. *Journal of Forestry*, 103, 2005.

[109] T. Winkler and B. Rinner. Applications of trusted computing in pervasive smart camera networks. In *Proceedings of the 4th Workshop on Embedded Systems Security*, page 2. ACM, 2009.

[110] T. Winkler and B. Rinner. TrustCAM: Security and Privacy-Protection for an Embedded Smart Camera based on Trusted Computing. In *Proceedings of the Conference on Advanced Video and Signal-Based Surveillance*, 2010.

[111] Xen. http://www.xen.org/, 2013.

[112] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 253–267, 2003.

